




데이터 정의와 설명

- [하이프의 데이터 타입 이해하기](#)
- [비라인에서의 하이브타입에 대한 연습](#)
- [데이터 타입 변환](#)
 - [목시적 타입 변환](#)
 - [명시적 타입 변환](#)
- [하이브 데이터 정의 언어\(DDL\)](#)
- [하이브 데이터베이스](#)
 - [하이브의 SHOW와 DESCRIBE](#)
- [하이브의 내부 및 외부 테이블](#)
 - [내부 테이블\(Internal Table\)](#)
 - [외부 테이블\(External table\)](#)
- [CTAS](#)
- [LIKE를 이용한 테이블 생성](#)
- [DROP TABLE / TRUNCATE TABLE](#)
 - [DROP TABLE](#)
 - [TRUNCATE TABLE](#)
- [CONCATENATE](#)
- [ALTER](#)
- [하이브 파티션](#)
- [하이브 버킷](#)
- [하이브 뷰](#)

하이브의 데이터 타입 이해하기

- 하이브의 데이터 타입 : 원시 타입과 복잡한 데이터 타입, 2 타입이 존재

원시 데이터 타입

|  원시 데이터 타입 |  설명 |  예제 |
|---|--|--|
| <u>TINYINT</u> | 1바이트이며 -128부터 127까지의 값을 가진다. 접미사는 Y. 작은 범위의 수를 표현할 때 사용한다. | 10Y |
| <u>SMALLINT</u> | 2바이트이며 -32,768부터 32,767까지의 값을 가진다. 접미사는 S. 일반적인 숫자를 표현하는 데 사용한다. | 10S |
| <u>INT</u> | 4바이트이며, -2,147,483,648에서 2,147,483,647 까지의 값을 가진다. | 10 |
| <u>BIGINT</u> | 8바이트이며, -9,223,372,036,854,775,808에서 9,223,372,036,854,775,807 까지의 값을 가짐. 접미사는 L이다. | 100L |
| <u>FLOATINT</u> | 4바이트 단일 정밀도 부동 소수점 숫자이다. 과학적 표기법은 아직 지원되지 않는다. 숫자값과 가장 가까운 근사치를 저장한다. | 1.23456789 |
| <u>DOUBLE</u> | 8바이트 배정밀도 부동 소수점 숫자이다. 과학적 표기법은 아직 지원되지 않는다. 숫자값과 가장 가까운 근사치를 저장한다. | 1.2345678901234567 |

| Aa 원시 데이터 타입 | ≡ 설명 | ≡ 예제 |
|------------------|--|---------------------------------|
| <u>DECIMAL</u> | 고정된 38자리 정확도를 가진 타입이다. 약 $(10^{39})-1 \sim (10^{38})$ 까지의 값을 가진다. DECIMAL 데이터 타입은 숫자의 정확한 표현을 저장한다. DEMICAL 데이터 타입의 기본 정의 값은 (10,0)이다. | 3.14와 같은 값을 저장하려면 DECIMAL (3,2) |
| <u>BINARY</u> | STRING의 CAST만 지원한다. | 1011 |
| <u>BOOLEAN</u> | TRUE 또는 FLASE 값이다. | TRUE |
| <u>STRING</u> | 작은따옴표(') 또는 큰따옴표(")로 표현되는 문자열을 포함한다. 하이브는 문자열에 C언어 스타일의 이스케이프 문자를 사용한다. 최대값은 약 2G이다. | 'BOOKS' 또는 "BOOKS" |
| <u>CHAR</u> | 작은따옴표(') 또는 큰따옴표(")로 표현되는 문자열을 포함한다. 최대 길이는 255로 고정되어 있다. | 'BOOKS' 또는 "BOOKS" |
| <u>VARCHAR</u> | 작은따옴표(') 또는 큰따옴표(")로 표현되는 문자열을 포함한다. 최대 길이는 65355로 고정되어 있다. 문자열이 VARCHAR타입으로 변환할 때 문자열의 길이가 너무 길면, 문자열에서 길이가 긴 부분은 잘려진다. | 'BOOKS' 또는 "BOOKS" |
| <u>DATE</u> | YYYY-MM-DD의 포맷으로 특정 년, 달, 일을 표현한다. 날짜의 범위는 0000-01-01에서 9999-12-31이다. | '2021-07-27' |
| <u>TIMESTAMP</u> | YYYY-MM-DD HH:MM:SS[.fff...]의 형식으로 특정 년, 달, 일, 시간, 분, 초, 밀리세컨드를 표현한다. | '2021-07-27' 19:44:13.345 |

원시 데이터 타입의 사본

| Aa 복잡한 데이터 타입 | ≡ 설명 | ≡ 예제 |
|---------------------|---|--------------------------------|
| <u>ARRAY</u> | [val1, val2, 등] 과 같이 값들이 모두 동일한 타입이다. array_name[index]를 사용(e.g. fruit[0]='apple')해 값에 접근할 수 있다. | ['apple','orange','mango'] |
| <u>MAP</u> | {key1:var1, key2:var2 등} 과 같이 키와 값을 짝으로 구성한 집합이다. map_name[key]를 사용(e.g. fruit[1]="apple")해 값에 접근할 수 있다. 이때, key:var이 int:string이라면 모든 key:var이 int:string의 형태여야 한다. | {1:"apple", 2:"orange"} |
| <u>STRUCT</u> | {var1, var2, var3, 등}과 같이 타입과 상관없이 필드를 사용자 정의한 구조이다. 즉, 값마다 타입이 다를 수 있고, 어떤 타입이든 가능하다. 기본적으로 STRUCT 필드 이름은 순서대로 col1, col2 등이 된다. structs_name.column_name을 사용(e.g. fruit.col1=1)해 값에 접근한다. | {1,"apple"} |
| <u>NAMED STRUCT</u> | {name1:var1, name2:var2, 등} 과 같이 타입을 가진 여러 필드를 사용자가 계속 정할 수 있는 구조다. structs_name.column_name을 사용(e.g. fruit.apple="gala")해 값에 접근할 수 있다. 이때, MAP과 달리 name:var의 쌍을 이룬 데이터 타입이 쌍마다 달라도 된다. | {"apple": "gala", "weight": 1} |
| <u>UNION</u> | 모든 데이터 타입 중 정확하게 하나의 데이터 타입만 갖는 구조다. 자주 사용되는 타입은 아니다. | {2:["apple","orange"]} |

- 복잡한 데이터 타입은 원시 데이터 타입을 기반으로 생성됨.
- ARRAY와 MAP은 자바의 타입과 동일하다.
- STRUCT는 테이블과 닮았다.
- 복잡한 데이터 타입은 타입의 중첩을 허용.

비라인에서의 하이브타입에 대한 연습

- mysql 및 hive 실행

```
sudo service mysql restart
./bin/hive
```

- 만약 beeline으로 실행시, HiveServer2 접속

```
./bin/beeline
beeline> !connect jdbc:hive2://localhost:10000 hduser 비밀번호
```

- 칼럼 이름(헤더) 출력 설정
 - 하이브 결과 집합에 있는 칼럼 이름은 항상 소문자

```
SET hive.cli.print.header = TRUE;
```

- 테이블 생성

```
CREATE TABLE employee(
  name string,
  work_place ARRAY<string>,
  sex_age STRUCT<sex:string,age:int>,
  skills_score MAP<string,int>,
  depart_title MAP<string,ARRAY<string>>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ',' // List 의 요소들을 구분하는 기준
MAP KEYS TERMINATED BY ':';        // MAP의 key와 value를 구분하는 기준
```

- 생성된 테이블의 칼럼 확인

```
desc employee;
```

```
hive> desc employee;
OK
name                string
work_place           array<string>
sex_age              struct<sex:string,age:int>
skills_score         map<string,int>
depart_title         map<string,array<string>>
Time taken: 0.119 seconds, Fetched: 5 row(s)
```

- 데이터 로드

```
load data local inpath '/usr/local/hadoop/employee.txt'
overwrite into table employee;
```

```
hive> select work_place from employee;
OK
work_place
["Montreal", "Toronto"]
["Montreal"]
["New York"]
["Vancouver"]
Time taken: 0.176 seconds, Fetched: 4 row(s)
```

- ARRAY format column 값에 접근
 - 해당 리스트의 index 값에 접근하여 출력

```
SELECT work_place[0] from employee;
```

```
hive> select work_place[0] as col1_workplace from employee;
OK
col1_workplace
Montreal
Montreal
New York
Vancouver
Time taken: 0.434 seconds, Fetched: 4 row(s)
hive> select work_place[1] as col2_workplace from employee;
OK
col2_workplace
Toronto
NULL
NULL
NULL
```

- NAMED_STRUCT format column 값에 접근
 - 칼럼명.키명 으로 접근 가능

```
SELECT sex_age.sex as sex, sex_age.age as age FROM employee;
```

```
hive> SELECT sex_age.sex as sex, sex_age.age as age FROM employee;
OK
sex      age
Male     30
Male     35
Female   27
Female   57
```

- MAP format column 값에 접근
 - 칼럼명["키명"]으로 접근 가능

```
SELECT skills_score["Python"] as python_score from employee;
```

```
hive> select skills_score["Python"] as python_score from employee;
OK
python_score
NULL
NULL
80
NULL
```

```
SELECT name, skills_score["DB"] as db, skills_score["Perl"] as perl, skills_score["Python"] as python,
skills_score["Sales"] as sales, skills_score["HR"] as hr
FROM employee;
```

| name | db | perl | python | sales | hr |
|---------|------|------|--------|-------|------|
| Michael | 80 | NULL | NULL | NULL | NULL |
| Will | NULL | 85 | NULL | NULL | NULL |
| Shelley | NULL | NULL | 80 | NULL | NULL |
| Lucy | NULL | NULL | NULL | 89 | 94 |

- Value format이 ARRAY인 MAP column에 접근
 - 전체 리스트 조회 : 칼럼명["칼럼"] 접근
 - 리스트의 특정 index 값 조회 : 칼럼명["칼럼"][index] 접근

```
SELECT depart_title["Test"][0] as test from employee;
```

```
hive> select depart_title["Test"][0] as test from employee;
OK
test
NULL
Lead
Lead
NULL
```

데이터 타입 변환

묵시적 타입 변환

- 좁은 타입에서 넓은 타입으로 변환. 넓은 타입에서 좁은 타입으로의 변환은 허락되지 않음.

- 모든 숫자 타입, FLOAT, STRING은 묵시적으로 DOUBLE, TINYINT, SMALLINT으로 변환될 수 있고, INT는 FLOAT으로 변환될 수 있음. BOOLEAN 타입은 기타 다른 타입으로 변환될 수 없다.

명시적 타입 변환

- CAST(value AS TYPE) 문법을 가진 CAST 함수를 사용.
- BINARY 타입은 STRING 으로만 변환 가능. 따라서 필요하다면 BINARY 타입을 STRING으로 변환한 후, 다른 타입으로 변환할 수 있음
- e.g. CAST('100' AS INT) 는 문자열 100을 숫자 값인 100으로 변환한 것.
- e.g. CAST('INT' AS INT) 와 같이 타입 변환이 실패하면, 해당 함수는 NULL 을 리턴.

하이프 데이터 정의 언어(DDL)

- 하이브 데이터 정의 언어(DDL, Data Definition Language)는 데이터베이스, 테이블, 뷰, 파티션, 버킷 같은 스키마 객체를 생성, 삭제, 변경함으로써 하이브의 데이터 구조를 설명하는 하이브 SQL 문의 부분 집합
- 대부분의 하이브 DDL 문은 CREATE, DROP, ALTER
- 하이브 DDL 문법은 SQL의 DDL 문법과 매우 유사. 하이브의 주석은 —으로 시작

하이프 데이터베이스

- 하이브의 데이터베이스는 비슷한 목적으로 사용되거나 동일한 그룹에 속한 테이블의 집합을 설명
- 데이터베이스를 명세하지 않으면 기본 데이터베이스가 사용됨. 기본 데이터베이스는 자신만의 디렉토리를 갖지 않는다.
- 새로운 데이터베이스가 생성될 때마다 하이브는 hive.metastore.warehouse.dir에 정의한 /user/hive/warehouse에 개별 데이터베이스를 위한 디렉토리를 생성한다.
 - 예를 들어, myhivebook 데이터베이스는 /user/hive/warehouse/myhivebook.db에 위치한다.
- 하이브는 데이터베이스와 테이블을 디렉토리 모드로 유지한다. 따라서 부모 디렉토리를 삭제하기 위해선 먼저 하위 디렉토리를 삭제해야 한다. 기본적으로 데이터베이스가 비어있거나 CASCADE를 명세하면, 영구히 삭제할 수 있다.
 - CASCADE는 데이터베이스를 삭제하기 전에 자동으로 데이터베이스의 테이블을 삭제한다.
- 디렉토리 존재 여부를 확인하지 않고, 데이터베이스를 생성

```
CREATE DATABASE myhivebook;
```

- 디렉토리 존재 여부를 확인한 후, 데이터베이스를 생성

```
CREATE DATABASE IF NOT EXISTS myhivebook;
```

- 저장 위치, 주석, 메타데이터 정보를 포함해 데이터베이스를 생성

```
CREATE DATABASE IF NOT EXISTS myhivebook
COMMENT 'hive database demo'
LOCATION '/hdfs/directory'
WITH DBPROPERTIES ('creator'='hyerim', 'date'='2021-08-01')
```

- 와일드카드로 데이터베이스를 보여줌

```
SHOW DATABASES LIKE 'my.*';
```

- 해당 데이터베이스의 describe를 출력

```
DESCRIBE DATABASE default;
```

- 데이터베이스를 사용

```
USE myhivebook;
```

- 빈 데이터베이스를 영구히 삭제

```
DROP DATABASE IF EXISTS myhivebook;
```

- CASCADE로 데이터베이스를 영구히 삭제

```
DROP DATABASE IF EXISTS myhivebook CASCADE;
```

- 데이터베이스 속성을 변경

```
ALTER DATABASE myhivebook
SET DBPROPERTIES ('edited_by'='Hyerim');
```

```
ALTER DATABASE myhivebook SET OWNER user hyerim;
```

하이프의 SHOW와 DESCRIBE

- 하이브의 SHOW와 DESCRIBE는 테이블, 파티션 등 하이브 객체의 대부분에 대한 정의 정보를 보여 주는 데 사용됨
- SHOW

- 테이블, 테이블의 속성, 테이블 DDL, 인덱스, 파티션, 칼럼, 함수, 잠금, 권한, 설정, 트랜잭션, 컴팩션(compaction)과 같은 하이브 객체의 넓은 범위를 지원
- DESCRIBE
 - 데이터베이스, 테이블, 뷰, 칼럼, 파티션과 같은 하이브 객체의 작은 범위를 지원.
 - 하지만, EXTENDED 또는 FORMATTED 키워드로 조합한 더 상세한 정보를 제공
 - DESC, DESCRIBE 모두 가능

하이브의 내부 및 외부 테이블

- 하이브의 테이블은 관계형 데이터베이스의 테이블과 매우 유사.

내부 테이블(Internal Table)

- 하이브에서 default로 테이블을 생성하면, internal table로 생성된다.
 - \$HIVE_HOME/conf/hive-site.xml에 설정된 디렉토리를 따라 테이블의 모든 데이터가 해당 디렉토리에 저장됨. 이렇게 하이브 디렉토리 안에 저장되는 데이터를 내부 테이블 또는 관리되는 테이블이라 부름.
 - 즉 테이블을 생성하면 HDFS와 Meta Store에 테이블을 저장한다.
 - 기본 디렉토리 : /user/hive/warehouse
 - e.g. 하이브에서 employee 테이블을 생성하면, HDFS에서 /user/hive/warehouse/employee 디렉토리가 생성됨.
- 삭제 : HDFS에 저장된 테이블과 메타데이터 모두 삭제됨

외부 테이블(External table)

- HDFS에 이미 데이터가 존재할때, HDFS에 존재하는 테이블의 위치를 명시하고 CREATE EXTERNAL TABLE 명령어를 통해 만든 테이블
 - 즉 테이블을 생성하면 Meta Store에 테이블을 저장.
 - 만약 LOCATION 속성 path의 폴더가 존재하지 않는다면, 하이브는 해당 폴더를 생성할 것.
 - LOCATION 속성에 명시된 폴더에 다른 폴더가 있다면, 하이브는 테이블을 생성할 때는 에러를 알리지 않지만, 테이블을 쿼리할 때는 에러를 알람
- 삭제 : 메타데이터만 삭제됨
- 외부 테이블 생성 방법

1. HDFS에 데이터를 저장

```
hdfs dfs -mkdir /sample // sample라는 디렉토리 안에 파일을 넣을 예정
hdfs dfs -put employee.txt /sample // sample 라는 디렉토리 안에 employee.txt 파일을 넣어줌
```

2. 외부 테이블 생성


```
CREATE EXTERNAL TABLE employee_external(
  name string,
  work_place ARRAY<string>,
  sex_age STRUCT<sex:string,age:int>,
  skills_score MAP<string,int>,
  depart_title MAP<string,ARRAY<string>>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
STORED AS TEXTFILE
LOCATION '/sample';
```

CTAS

- 테이블 정의뿐 아니라 기존 테이블 값을 복사하여 새로운 테이블을 생성하는 것. 즉 테이블 생성과 데이터 로드가 동시에 이루어짐.
- CTAS에 의해 생성된 테이블은 원자적이다. 즉, 다른 사용자는 쿼리 결과가 나타날 때까지 테이블을 보지 못한다
- CTAS의 제한
 - CTAS로 생성된 테이블은 파티션 테이블이 될 수 없다.
 - CTAS로 생성된 테이블은 외부 테이블이 될 수 없다.
 - CTAS로 생성된 테이블은 목록 버킷 테이블이 될 수 없다.

```
CREATE TABLE ctas_employee
AS SELECT * FROM employee_external;
```

- CTE(Common Table Expression)이 포함된 CTAS 생성
 - CTE
 - WITH 절 뒤에 명시된 간단한 SELECT 쿼리 결과를 제어하기 위해 SELECT 또는 INSERT 키워드가 추가된 임시 결과 집합.
 - 하나의 문장의 실행 범위에서만 정의됨
 - 하나 이상의 CTE는 SELECT, INSERT, CREATE TABLE AS SELECT, CREATE VIEW AS SELECT 문과 같은 하이브 키워드를 포함해 중첩 또는 체인 방식으로 사용됨

```
CREATE TABLE cte_employee AS
WITH r1 AS
(SELECT name FROM r2 WHERE name='Michael'),
r2 AS
(SELECT name FROM employee WHERE sex_age.sex='Male'),
r3 AS
(SELECT name FROM employee WHERE sex_age.sex='Female')
SELECT * FROM r1 UNION ALL SELECT * FROM r3;
```

```
# 생성된 테이블 확인
SELECT * FROM cte_employee;
```



LIKE를 이용한 테이블 생성

- like 뒤에 테이블과 같은 컬럼 형식의 테이블이 생성됨
- 테이블의 데이터가 그대로 복사되진 않음

```
CREATE TABLE empty_like_employee
LIKE employee_internal;
```

DROP TABLE / TRUNCATE TABLE

DROP TABLE

- 메타데이터를 완벽히 삭제하고 데이터를 Trash로 이동.
- 만약 Trash가 설정됐다면, 설정된 Trash 디렉토리로 데이터를 이동

TRUNCATE TABLE

- 테이블의 모든 로우를 삭제
- 해당 테이블은 내부 테이블이어야 함

```
TRUNCATE TABLE employee;
```

CONCATENATE

- RCFile을 블록 레벨로 빠르게 병합하거나, ORC 파일을 스트라이프 레벨로 빠르게 병합하기 위한 기능
- 압축을 해제하거나 데이터를 디코딩하는 오버헤드를 피할 수 있다.
- 테이블의 이름을 변경

```
ALTER TABLE apache_access_logs RENAME TO empty_table;
```

- 테이블의 주석 속성을 변경

```
ALTER TABLE empty_table
SET TBLPROPERTIES ('commnet'='empty, apache access logs')
```

- 테이블의 구분자를 변경

```
ALTER TABLE empty_table
SET SERDEPROPERTIES ('field.delim'='$');
```

- 테이블의 파일 포맷을 변경

```
ALTER TABLE empty_table
SET FILEFORMAT RCFILE;
```

- HDFS의 전체 URI로 설정해야 할 테이블의 장소를 변경

```
ALTER TABLE empty_table
SET LOCATION
'hdfs://DESKTOP-SJTS912:9010/sample';
```

- 테이블의 enable/disable 보호 정책을 NO DROP으로 변경
 - NO_DROP : 데이터 삭제를 방지
 - OFFLINE : 쿼리로부터 테이블의 데이터를 보호(메타데이터는 보호하지 않음)

```
ALTER TABLE empty_table ENABLE NO_DROP; // 삭제 방지
ALTER TABLE empty_table DISABLE NO_DROP; // 삭제 허용
ALTER TABLE empty_table ENABLE OFFLINE; // 쿼리로부터 보호
ALTER TABLE empty_Table DISALBE OFFLINE; // 쿼리로부터 보호 해제
```

- 작은 파일을 큰 파일로 병합

```
# 지원하는 파일 포맷으로 변환
ALTER TABLE empty_table SET FILEFORMAT ORC;

# 파일에 CONCATENATE를 실행
ALTER TABLE empty_table CONCATENATE;

# 일반 파일 포맷으로 변환
ALTER TABLE empty_table SET FILEFORMAT TEXTFILE;
```

ALTER

- 칼럼의 데이터 타입을 확인
- 메타데이터만 변경하고, HDFS 파일은 변형시키지 않음.
- 칼럼의 데이터 타입 변경

```
DESC employee;
```

- 칼럼명 변경
 - ALTER TABLE <변경할 테이블명> CHANGE <OLD칼럼명> <NEW칼럼명> <칼럼타입>

```
ALTER TABLE like_employee  
CHANGE name employee_name
```

- 칼럼 추가
 - ALTER TABLE <변경할 테이블명> ADD COLUMNS (<칼럼명> <칼럼타입>)

```
ALTER TABLE like_employee  
ADD COLUMNS (new_name string);
```

- 칼럼을 치환
 - ALTER TABLE <변경할 테이블명> REPLACE COLUMNS (<남길 칼럼명> <남길 칼럼의 타입>)
 - REPLACE COLUMNS 뒤에 남길 칼럼명을 적으면, 적힌 칼럼 외에 칼럼들이 모두 삭제됨.

```
ALTER TABLE like_employee  
REPLACE COLUMNS (name string, work_place array<string>);
```

하이프 파티션

- 하이브의 간단한 쿼리는 모든 하이브 테이블을 스캔. 따라서 큰 데이터를 쿼리할 때, 성능저하가 발생.
- 하이브 파티션을 생성하면 성능저하 이슈를 해결할 수 있음. 테이블이 쿼리를 받으면, 테이블 데이터 중 필요한 파티션(디렉토리)만 쿼리되기 때문에, I/O와 쿼리 시간을 많이 줄일 수 있음
- 하이브 파티션은 RDBMS의 파티션과 매우 유사. 각 파티션은 미리 정의된 파티션 칼럼 값에 해당.
- **파티션마다 HDFS 테이블의 디렉토리에 하위 디렉토리가 생성되고, 하위 디렉토리에 해당 파티션이 저장됨.**
 - 따라서 cardinality가 낮은 경우는 괜찮지만 cardinality가 높은 경우, 파티션을 사용하면 너무 많은 디렉토리가 생기게 됨 → 버킷 사용
- 테이블을 생성할 때 하이브 파티션을 생성하면 됨

```
CREATE TABLE employee_partitioned(  
name string,  
work_place ARRAY<STRING>,  
sex_age STRUCT<sex:string,age:int>,  
skills_score MAP<string,int>,  
depart_title MAP<string,ARRAY<string>>  
)
```

```
PARTITIONED BY (Year Int, Month Int)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';
```

- 테이블에 파티션을 추가
 - ALTER TABLE ADD PARTITION 사용
 - ADD PARTITION은 테이블의 메타데이터를 변경하지만 데이터를 바로 읽지 않음. 데이터가 파티션의 위치에 없다면, 쿼리는 어떤 결과도 리턴하지 않음

```
ALTER TABLE employee_partitioned
ADD PARTITION (year=2014, month=11)
PARTITION (year=2014, month=12);
```

- 파티션 확인

```
SHOW PARTITIONS employee_partitioned;
```

- 테이블에서 파티션을 삭제
 - 데이터와 메타데이터를 포함해서 파티션을 삭제하려면 ALTER TABLE <테이블명> DROP PARTITION 사용

```
ALTER TABLE employee_partitioned
DROP IF EXISTS PARTITION (year=2014, month=12);
```

- 데이터를 로딩하는 3가지 방법에 파티션을 명시해주면 됨.
 1. LOAD DATA : 로컬 파일 혹은 HDFS 파일을 로딩(data loading)
 2. INSERT INTO TABLE : 기존 데이터를 덮어쓰지 않고 데이터를 입력(adding data)
 3. INSERT OVERWRITE : 데이터를 입력할 때 기존 데이터를 덮어씀(replacing data)
- 파티션 칼럼이 한개일 때 예시

```
INSERT OVERWRITE TABLE iris2
PARTITION (Species = 'setosa')
SELECT Id, Species
From iris
WHERE iris.Species='setosa';
```

```
LOAD DATA LOCAL INPUT '/usr/local/hadoop/iris.csv'
OVERWRITE INTO TABLE iris2
PARTITION (Species = 'setosa')
SELECT Id, Species
From iris
WHERE iris.Species='setosa';
```

- 파티션 칼럼이 두개일 때 예시

```
INSERT INTO TABLE iris2
PARTITION (Species = 'setosa', SepalLength = 2)
SELECT Id, Species, SepalLength
from iris
WHERE iris.Species='setosa' AND SepalLength = 2
```

- ALTER TABLE의 구문에 PARTITION을 명시해주면 됨
 - ALTER TABLE <테이블명> PARTITION (<파티션>) ~~
 - ~~
 - SET FILEFORMAT
 - SET LOCATION
 - ENABLE
 - DISABLE
 - CONCATENATE

하이프 버킷

- 파티션 외에도 버킷(bucket)은 쿼리 성능을 최적화하기 위해 데이터 집합을 관리할 수 있는 작은 부분으로 잘게 나누는 또 다른 기술
- 파티션과 달리 버킷은 HDFS의 파일 세그먼트에 해당. 즉, HDFS의 파일 세그먼트로 저장.
 - 따라서 cardinality가 높은 칼럼의 경우에도 용이
- 버킷 개수
 - 버킷은 버킷 개수를 고정할 수 있음.
 - 적절한 버킷 개수를 정의하려면 각 버킷에 너무 많거나 적은 데이터를 사용하지 않아야함. 즉, 각 버킷간에 고르게 데이터를 분포하는 것이 좋음
 - 두 블록으로 설정하는 것도 좋음. 예를 들어 하둡의 블록 크기가 256MB이라면 각 버킷에 512MB를 계획할 수 있음
 - 가능하다면 2^N 을 버킷 개수로 사용
- 버킷을 이용해 맵 사이드 조인과 샘플링을 쉽고 효율적으로 수행할 수 있음.
- 이미 HDFS에 파일이 존재해야 함. 따라서 LOAD DATA LOCAL INPATH를 사용할 수 없고, INSERT로 데이터 로드 가능

```
# 테이블 생성
CREATE TABLE employee_id(
name string,
employee_id int,
work_place ARRAY<STRING>,
sex_age STRUCT<sex:string,age:int>,
```

```
skills_score MAP<string,int>,
depart_title MAP<string,ARRAY<string>>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';

# 테이블 로드
LOAD DATA LOCAL INPATH '/usr/local/hadoop/employee_id.txt'
OVERWRITE INTO TABLE employee_id;

# 버킷 테이블 생성
CREATE TABLE employee_id_buckets(
name string,
employee_id int,
work_place ARRAY<STRING>,
sex_age STRUCT<sex:string,age:int>,
skills_score MAP<string,int>,
depart_title MAP<string,ARRAY<string>>
)
CLUSTERED BY (employee_id) INTO 2 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';
```

- 버킷 개수를 강제로 적용

```
SET map.reduce.tasks=2;
```

```
SET hive.enforce.bucketing=true;
```

- 버킷 테이블에 데이터 로드

```
INSERT OVERWRITE TABLE employee_id_buckets
SELECT * FROM employee_id;
```

- 버킷 생성 확인

- 0000nn_0파일은 모두 버킷을 나타냄

```
hdfs dfs -ls /user/hive/warehouse/employee_id_buckets;
```

- 쿼리문에서 버킷 데이터 샘플을 사용하는 방법 : TABLESAMPLE
 - 3개의 버킷 중 첫 번째 버킷에서 샘플을 조회

```
SELECT employee_id
FROM employee_id_buckets
TABLESAMPLE(BUCKET 2 OUT OF 2);
```

하이프 뷰

- 하이브의 뷰는 join, subquery, 데이터 필터, 데이터 평탄화와 같은 복잡성을 숨겨서 쿼리를 간단히 만들기 위해 사용되는 논리적인 데이터 구조.
- 일부 RDBMS와는 다르게, 하이브 뷰는 데이터를 저장하거나 실체화된 데이터가 없다.
- 하이브 뷰가 생성되자마자 뷰의 스키마가 즉시 고정된다. 이후 기본 테이블의 변경한 내용(e.g. 칼럼 추가)이 뷰의 스키마에 반영되지 않음.
- 기본 테이블이 삭제되고 변경되면, 인식 불가능한 뷰의 모든 쿼리 시도는 실패
- 뷰를 생성할 때, 메타데이터만 변경되기 때문에 맵리듀스 작업은 전혀 실행되지 않음. 하지만 맵리듀스 잡이 실행되어야 하는 쿼리였다면, 뷰를 쿼리할 때 적절한 맵리듀스 잡이 실행될 수 있음
 - 예를 들어, create view count_man_employee as select count(*) from employee where sex_age.sex='Male' 로 뷰를 생성했다면, 생성 시에는 맵리듀스 잡이 실행되지 않음. 하지만 select * from count_man_employee와 같이 쿼리 시에는 맵리듀스 잡이 실행됨.

```
# 뷰 생성
CREATE VIEW employee_skills
AS
SELECT name, skills_score['DB'] AS DB,
skills_score['Perl'] AS Perl,
skills_score['Python'] AS Python,
skills_score['Sales'] AS Sales,
skills_score['HR'] AS HR
FROM employee;
```

- 뷰의 속성을 변경

```
ALTER VIEW employee_skills
SET TBLPROPERTIES ('comment'='This is View');
```

- 뷰를 재정의

```
ALTER VIEW employee_skills
AS
SELECT * FROM employee;
```

- 뷰를 삭제

```
DROP VIEW employee_skills;
```