

1. SECTION1 中 Introduction

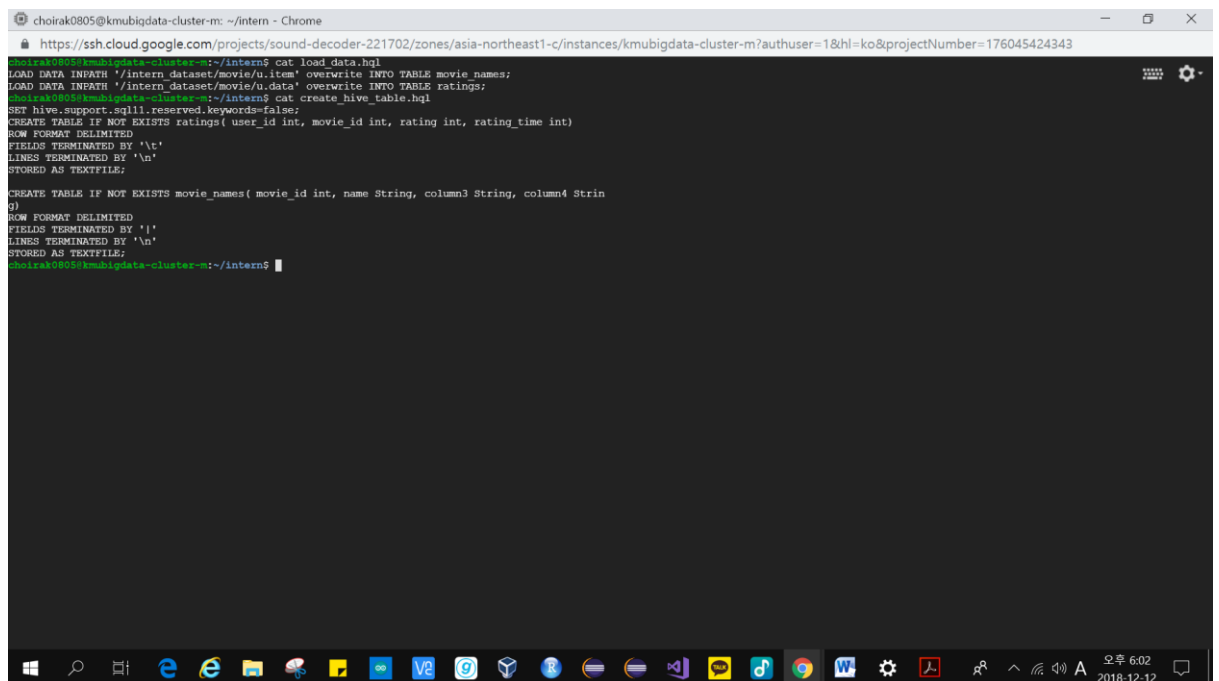
- HADOOP에서 추상화를 통해 서버 몇 개 쓰는지 환경 설정할 필요 없다.
- 실습: *SANDBOX CONFIGURATION* 과정에 *ERROR* 많아서 익숙한 *GOOGLE CLOUD* 사용

2. SECTION2 中 HIVE 체험 및 MAP-REDUCE

- LOCAL에 있는 FILE을 HDFS에 옮겨서 진행해야 함.
- 우선, HIVE에 FILE 넣기 위해 SETTING 진행한다.

.hql 파일을 만들어야 한다. LOCAL에서 'CREATE TABLE. hql'을 만들고 그것을 HIVE 명령어로 실행해서 HIVE에서 테이블이 생성되게 한다. 그리고 HDFS에 실제 DATA를 넣어놓고 해당 DATA를 HIVE TABLE에 OVERWRITE한다.

1. `hdfs dfs -mkdir -p /intern_dataset/movie`
2. `hdfs dfs -put u.data /intern_dataset/movie` //HDFS에 LOCAL파일 올린다.
3. `hdfs dfs -put u.item /intern_dataset/movie`
4. `hive -f 실행할파일.hql`



```
choirak0805@kmubigdata-cluster-m: ~/intern - Chrome
https://ssh.cloud.google.com/projects/sound-decoder-221702/zones/asia-northeast1-c/instances/kmubigdata-cluster-m?authuser=1&hl=ko&projectNumber=176045424343

~/intern$ cat load_data.hql
LOAD DATA INPATH '/intern_dataset/movie/u.items' overwrite INTO TABLE movie_names;
LOAD DATA INPATH '/intern_dataset/movie/u.data' overwrite INTO TABLE ratings;
~/intern$ cat create_hive_table.hql
SET hive.support.sql1.reserved.keywords=false;
CREATE TABLE IF NOT EXISTS ratings(user_id int, movie_id int, rating int, rating_time int)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

CREATE TABLE IF NOT EXISTS movie_names(movie_id int, name String, column3 String, column4 String)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

~/intern$
```

→ SQL 실습 진행.

3. OVER VIEW OF HADOOP

→ HADOP이란 클러스터의 전체 컴퓨터에서 동작하며 분산 저장한 DATA에 대한 SW.

단순히 PC 추가해서 용량 늘릴 수 있고 SINGLE FILE로 PROCESS까지 진행 할 수 있다.

이 때 PROCESS과정은 PARALLEL하게 진행된다. AWS나 GCP 등을 통해 HW 빌려서 진행 할 수 있다.

→ GFS는 BIG-DATA 저장용 MR은 BIG-DATA PROCESSING과정이다.

→ FAULT-TOLERANCE. REPLICATION을 통해 해결한다.

→ HORIZONTAL SCALE이다. SCALE-OUT.

→ **HADOOP ECOSYSTEM**

CORE HADOOP ECOSYSTEM: HDFS (DATA STORAGE), YARN (MANAGE RESOURCE인데 MESOS랑 비슷하다.), MAP-REDUCE (PROGRAMMING MODEL. MAPPER: TRANSFORM PARALLEL REDUCER: AGGREGATE), PIG (SUPPORT SCRIPT LANGUAGE. TRANSFORM SCRIPT TO MR LANGUAGE), HIVE(NON RELATIONAL에 대해서도 TAKE SQL QUERY AND PROCESS AS MR), AMBARI(VISUALIZE ALL OF CLUSTER PROCESS, EXECUTE, IMPORT → VIEW로 모든 상황을 보여준다). SPARK (MR과 같다. SPARK SCRIPT를 SCALA, PYTHON 등으로 쓸 수 있다. 동작이 빠르다. IN-MEMORY다. 머신러닝도 할 수 있다.)HBASE(NOSQL DATABASE랑 같다. COLUMNAL DATA다. FAST다. MR이나 SPARK 작업 결과를 빠르게 쓸 수 있다).

EXTERNAL DATA STORAGE: MYSQL(CUSTER에서의 DATA에 WRITE, READ가능하다), MONGODB(COLUMNARL DATABASE.REAL TIME 과 CLUSTER 사이에 사용한다. KEY-VALUE 형식으로 사용된다.)

4. HDFS

→ BIG DATA STORAGE.

→ 큰 파일을 BLOCK으로 나눈다. 128MB단위로 나눈다.

→ 몇 개의 컴퓨터들이 다른 BLOCK들에 PARALLEL하게 접근해서 PROCESSING가능하다.

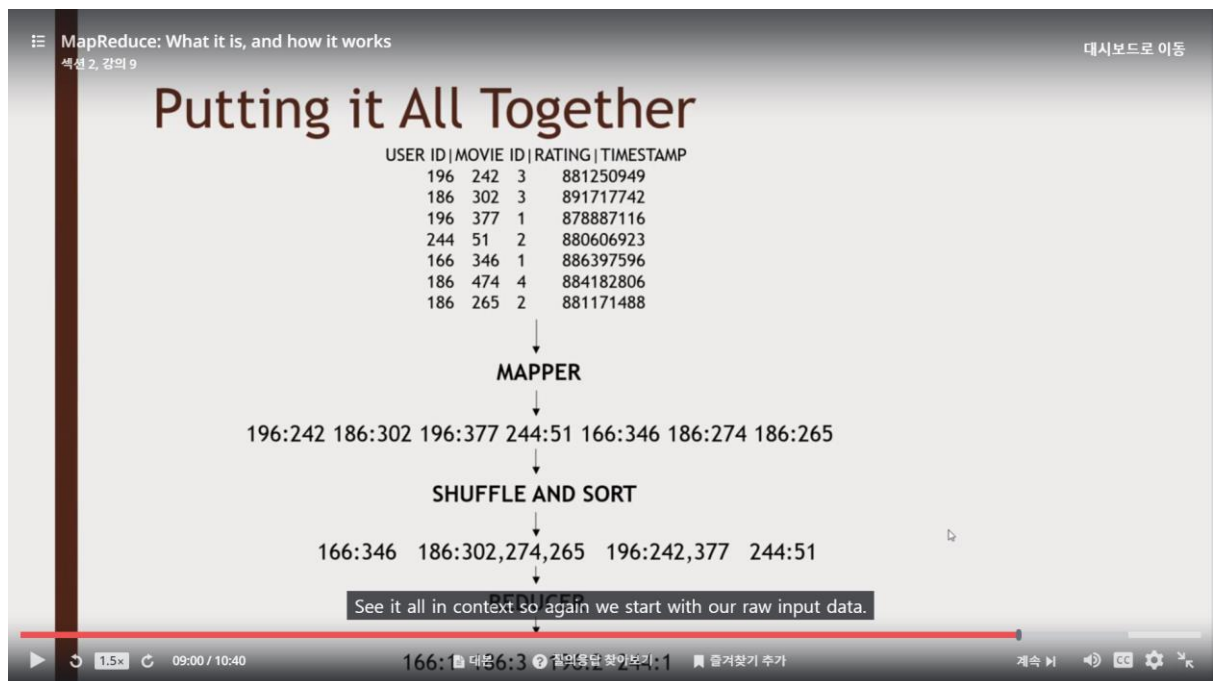
- ➔ FAILURE 때문에 3개의 REPLICATION을 가지고 HDFS가 FAILURE를 다룬다.
- ➔ SCALE OUT이라서 FAILURE일어나도 REPLICATION이용한다.
- ➔ NAMENODE(METADATA를 가지고 테이블을 가지고 파일에 대한 모든 BLOCK정보와 모든 작업 정보를 가진다. DATA NODE에 대한 모든 정보를 가진다. DATA NODE는 실제 BLOCK들을 저장하고 있다.
- ➔ CLIENT NODE에서 HDFS의 READ한다면 NAME NODE에게 읽고 싶은 파일을 물어보고 NAME NODE가 효율적으로 어떻게 읽을 수 있는지 어느 DATA NODE에 있는지 알려준다.
- ➔ WRITE할 때는 CLIENT가 NAME NODE에게 저장한다고 말한다. 그리고 CLIENT는 DATA NODE에게 말하면 DATA NODE 간의 REPLICATION과 저장을 한다. 그리고 DATA NODE는 저장 결과를 NAME NODE에게 저장 정보를 알려준다.
- ➔ NAME NODE FAULT-TOLERANCE.
NAME NODE:
-BACK UP METADATA, NAME NODE FAILURE일어나도 META DATA LOG따라서 복구 가능하다.
-SECONDARY NAME NODE는 모든 LOG에 대한 통합된 COPY를 가지고 있어서 이것으로 복구 가능하다. 이게 METADATA보다 낫다.
- ➔ 실제 HDFS에 파일 올리기: AMBARI를 통해 HTTP 로 HDFS 간편하게 볼 수 있다. 간단하게 SANDBOX에 UI로 올릴 수 있고 파일 탐색도 간편하다. 이 때 DATA NODE에 알아서 REPLICATION되면서 저장 될 것이다. 파일업로드/다운로드 간편하게 된다.
DATA NODE에 여러 BLOCK으로 저장되지만 하나의 파일로 볼 수 있다.
- ➔ PUTTY를 통해서 SSH 통신으로 CLUSTER에 접근 가능하다. 이것을 통해서 HDFS에 파일을 넣으려면 maria_dev@17.0.0.1를 ip에 쓰고 port에 2222 작성한다. 그리고 명령어를 통해서 접근한다. 나는 `hdfs dfs -put 데이터 /Hadoop path`를 사용했지만 여기서는 `Hadoop fs -put ~`을 사용함.
- ⇒ 정리하자면 UI를 통해서 HDFS에 접근할 수 있다. 하지만 CLI로도 가능한데 둘 다 모두 HDFS에 PUT하거나 LS로 탐색할 수 있다.

5. MAP-REDUCE

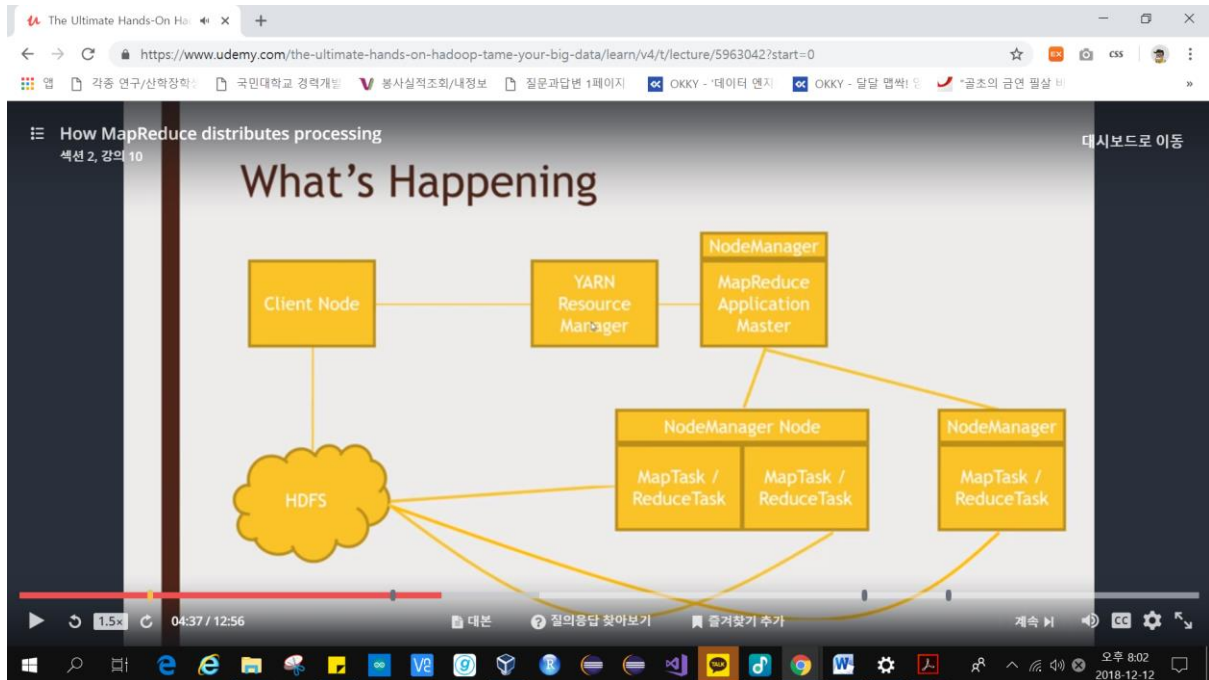
- DISTRIBUTING PROCESSING 목적. PARALLEL하게 적용된다.
- MAPPER는 DATA TRANSFORM 목적. REDUCER는 AGGREGATE 목적이다.
- MR작업 간 KEY-VALUE로 작업된다.
- MAPPER는 LINE 별로 INPUT을 받아서 작업한다. KEY-VALUE로 RETURN한다.

이 때, RECORD에서 (BLOCK을 LINE 단위로 RECORD 별 작업하는데 필요한 정보를 KEY-VALUE로 RETURN하고 나머지는 WASTE)←NETWORK에 최소한의 영향 주도록 한다.

- REDUCER는 KEY-VALUE로 받아서 AGGREGATE한다. 이 때 MAP의 결과는 PARTITIONER에 의해 SHUFFLE된다(KEY기준으로 묶이도록 SHUFFLE된다). 그리고 KEY를 기준으로 SORT되어서 REDUCER의 결과로 들어간다.
- MAP-REDUCE과정



- SHUFFLE & SORT과정에서는 별 다른 개발자가 따로 PROGRAMMING할 필요 없다.
- MAPPER때와 같이 REDUCER또한 PARALLEL하게 진행된다.



➔ CLIENT가 접근하면 먼저 YARN에게 작업 할 수 있는 PC있는지 알려준다. 그리고 그와 동시에 DATA를 HDFS에 COPY한다. 그리고 NODE MANAGER는 NODE관리하는데 NODE MANAGER가 MR작업을 DISTRIBUTE한다 이 과정에서 필요한 DATA와 OUTPUT은 HDFS랑 상호 작용한다

➔ 이 때 NODEMANAGER는 MR 작업 할 NODE를 DATA랑 가장 가까운 NODE에서 작업하고자 한다. 이 때 DATA는 BLOCK으로 쪼개져 있는 상태다.

➔ MR은 거의 JAR파일로 진행된다. PYTHON 같은 것도 가능하다.

➔ HANDGLING FAILURE :

MASTER NODE경우 YARN이 다른 걸로 RESTART한다.

전체 NODE 경우는 RESOURCE MANAGER가 전체 NODE를 RESTART한다.

RESOURCE MANAGER의 경우는 ZOOKEEPER에게 NODE들이 RESOURCE MANAGER를 새로 할당 받으면 된다.

➔ COMBINER의 경우는 MAPPER NODE들 결과에 대해서 REDUCTION하는 것인데 OVERHEAD존재한다. 예를 들어) WORD COUNT에서 <A, ONE> <A, ONE>일 때 <A, TWO>로 REDUER에 들어가게 COMBINE한다.

➔ **실습: MOVIE RATING에서 RATING COUNT**

실제로 RAW MR작업보다는 SQL을 이용하는 것이 편할 때도 있다.

The Ultimate Hands-On Hadoop: Tame Your Big Data

https://www.udemy.com/the-ultimate-hands-on-hadoop-tame-your-big-data/learn/v4/t/lecture/5963046?start=0

MapReduce example: Break down movie ratings by rating score

Making it a MapReduce problem

- MAP each input line to (rating, 1)
- REDUCE each rating with the sum of all the 1's

USER ID	MOVIE ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

Map: (rating, 1)

Shuffle & Sort: 1 -> 1, 1; 2 -> 1, 1; 3 -> 1, 1; 4 -> 1

Reduce: 1, 2; 2, 2; 3, 2; 4, 1

It's just each line contains a specific rating and each column of that rating contains a userID and a

MAPPER에서는 필요한 정보만 뺀다. RATING정보만 사용할 것이다.

<RATING(점수), 1> 그리고 이걸 SHUFFLE&SORT해서 REDUCER에서 RATING별 1들을 AGGREGAT해서 COUNT한다

--PYTHON CODE--

```
def mapper_get_ratings(self, _, line):
    (userID, movieID, rating, timestamp) = line.split('\t')
    yield rating, 1
```

(_, LINE) 입력 KEY-VALUE다.

그리고 읽은 VALUE인 LINE을 TAB 기준으로 SPLIT해서 각 변수에 넣는다.

그리고 <RATING, 1> RETURN한다.

```
def reducer_count_ratings(self, key, values):
    yield key, sum(values)
```

KEY-VALUE를 받아서 INTERABLE한 VALUE들을 SUM해서 <KEY, SUM(VALUE)> 출력

Putting it all together

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class RatingsBreakdown(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_ratings,
                  reducer=self.reducer_count_ratings)
        ]

    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield rating, 1

    def reducer_count_ratings(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    RatingsBreakdown.run()
```

- ➔ MRJOB, MRSTEP IMPORT해서 간편하다.
- ➔ RATINGBREAKDOWN이라는 작업이 있으면 어떤 것이 진행될 거인지 STEP에서 정의하고 작업들에 대한 함수를 정의한다.
- ➔ CLASS RATINGSBREAKDOWN(MRJOB)은 MRJOB을 CLASS가 INHERIT한다는 것이다.
- ➔ PYTHON CODE HADDOP에서 돌리기:

Running with mrjob

- Run locally
 - `python RatingsBreakdown.py u.data`
- Run with Hadoop
 - `python MostPopularMovie.py -r hadoop --hadoop-streaming-jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar u.data`

Now obviously if you're running just on your own PC you

- ➔ Python 실행파일 `-r hadoop --hadoop-streaming-jar [jar파일경로] [데이터파일경로]`
ex) 데이터파일경로를 `hdfs://~~` 형식으로 되어있을 수 있다.
- ➔ 또한 일반적으로 CLUSTER에서 전체 DATASET을 돌리기 전에, DATA의 일부분만 LOCAL COMPUTER에서 돌려보면서 TEST할 수 있다.
- ➔ **2개의 MR작업을 이용해서 MAP -> REDUCER로 갈 때 KEY 기준 SORT되는 것을 이용하여 VALUE를 KEY로 바꿔줘서 VALUE기준 SORT되도록 한다.**



- ➔ 가장 많이 나온 MOVIE 순으로 출력. MOVIE COUNT의 SUM을 KEY로 두 번째 KEY로 들어오면서 가장 먼저 출력된다. 두 번째 REDUCER에서는 같은 COUNT를 가진 영화들의 LIST로 들어올 수 있어서 FOR문으로 YIELD한 것이다.

3. SECTION5 中 HIVE

- ➔ HADOOP 에서의 SQL지원한다. SQL을 MR로 바꿔서 실행한다.
- ➔ EASY OLAP QUERIES. HIVE SQL과 같은 역할의 MR작업을 할 수 있다. 하지만 편리성을 제공해준다.
- ➔ NOT APPROPRIATE FOR OLTP(HIGH THROUGHPUT, LOW LATENCY필요한 경우)
- ➔ 실제로는 DE-NORMALIZED되어 있다.

➔ NO RECORD-LEVEL UPDATE, INSERT, DELETE

The screenshot shows a Hive query execution interface. The top section is a 'Worksheet' containing a SQL query:

```
1 CREATE VIEW topMovieIDs AS
2 SELECT movieID, count(movieID) as ratingCount
3 FROM ratings
4 GROUP BY movieID
5 ORDER BY ratingCount DESC;
6
7 SELECT n.title, ratingCount
8 FROM topMovieIDs t JOIN names n ON t.movieID = n.movieID;
```

Below the query, there are buttons for 'Execute', 'Explain', 'Save as...', and 'New Worksheet'. The bottom section is titled 'Query Process Results (Status: SUCCEEDED)' and shows the results of the query. It includes a 'Filter columns...' input field and 'previous' and 'next' buttons. The results are displayed in a table with two columns: 'n.title' and 'ratingcount'.

n.title	ratingcount
Star Wars (1977)	583
Contact (1997)	509
Fargo (1996)	508

➔ RATING많이 된 영화 이름 순으로 출력한다.

Schema On Read

- Hive maintains a “metastore” that imparts a structure you define on the unstructured data that is stored on HDFS etc.

```
CREATE TABLE ratings (
  userID INT,
  movieID INT,
  rating INT,
  time INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '${env:HOME}/ml-100k/u.data'
OVERWRITE INTO TABLE ratings;
```

➔ 실제 SCHEMA에서 정의하는 구조로 TABLE이 저장되는 것이 아니라 단순히 DELIMITED된 TEXTFILE형식 이런 식으로 저장되어 있다. HIVE가 METASTORE가 있는데 이것이 어떤 구조로 RAW DATA를 HDFS로 읽어 들일 것인지 정보를 가진다.

Where is the data?

- LOAD DATA
 - *MOVES data from a distributed filesystem into Hive*
- LOAD DATA LOCAL
 - *COPIES data from your local filesystem into Hive*
- Managed vs. External tables

```
CREATE EXTERNAL TABLE IF NOT EXISTS ratings (  
    userID INT,  
    movieID INT,  
    rating INT,  
    time INT)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'  
LOCATION '/data/ml-100k/u.data';
```

- ➔ LOAD DATA는 HDFS의 FILE을 HIVE로 MOVE시킨다. LOAD DATA LOCAL은 LOCAL의 DATA를 HIVE로 옮긴다.
- ➔ MANAGED와 달리 EXTERNAL은 DATA를 해당 위치에 그대로 넣어놔서 다른 SYSTEM과 SHARE할 수 있게 한다.
- ➔ PARTITIONING: 보통의 QUERY가 특정 값에 대한 거라면 그것에 대한 PARTITION을 만들면 된다.

REFERENCE: <http://paulsmooth.tistory.com/133>

Hive Partitioning

개요

- Hive 파티셔닝의 개념은 RDBMS 와 크게 다르지 않다. 테이블을 하나 이상의 키로 파티셔닝 할 수 있으며, 이것은 각 테이블에 데이터가 얼마나 저장될 것이냐를 기준으로 설정하면 된다. 예를 들어 테이블이 id, name, age 3개의 칼럼으로 구성되어 있고 age로 파티셔닝 하기로 설정하였더라면, 같은 나이를 갖는 row 들이 물리적으로 같이 저장된다.

파티션 테이블

- 일반적으로 non-partition 테이블은 아래와 같이 선언할 수 있다.

```
create table salesdata_source(  
    salesperson_id int,  
    product_id int,  
    date_of_sale string  
)
```

- 이와 같은 구조를 'data_of_sale' 로 아래와 같이 파티션 할 수 있다.

```
create table salesdata (  
    salesperson_id int,  
    product_id int  
)partitioned by (date_of_sale string)
```

```
CREATE VIEW IF NOT EXISTS avgRatings AS  
SELECT movieID, AVG(rating) as avgRating, COUNT(movieID) as ratingCount  
FROM ratings  
GROUP BY movieID  
ORDER BY avgRating DESC;  
  
SELECT n.title, avgRating  
FROM avgRatings t JOIN names n ON t.movieID = n.movieID  
WHERE ratingCount > 10;
```

➔ MOVIEID별로 평균 RATE와 빈도를 VIEW로 만든다. 그리고 10번 넘게 RATING된 것에 대해서 영화이름과 평균RATE를 출력한다.

➔ MYSQL을 SQOOP을 통해서 HADOOP CLUSTER에서 사용 가능하다.

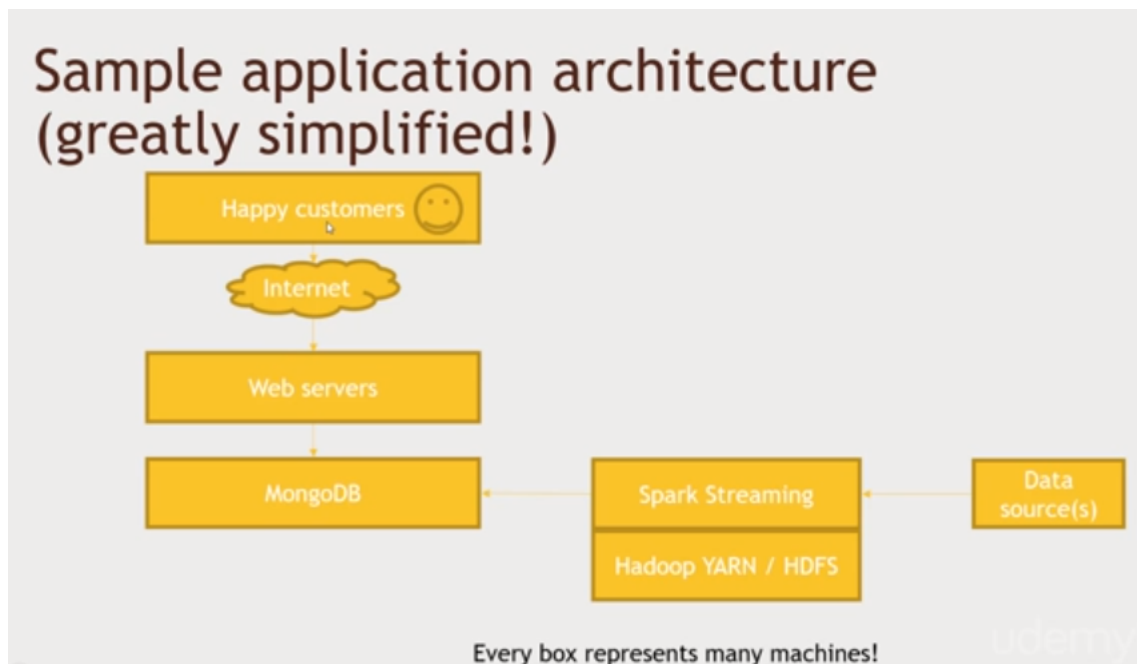
SQOOP을 통해서 MR작업 형태로 IMPORT / EXEPORT가능하다.

(CONFIGURATION하는 과정이라 SKIP)

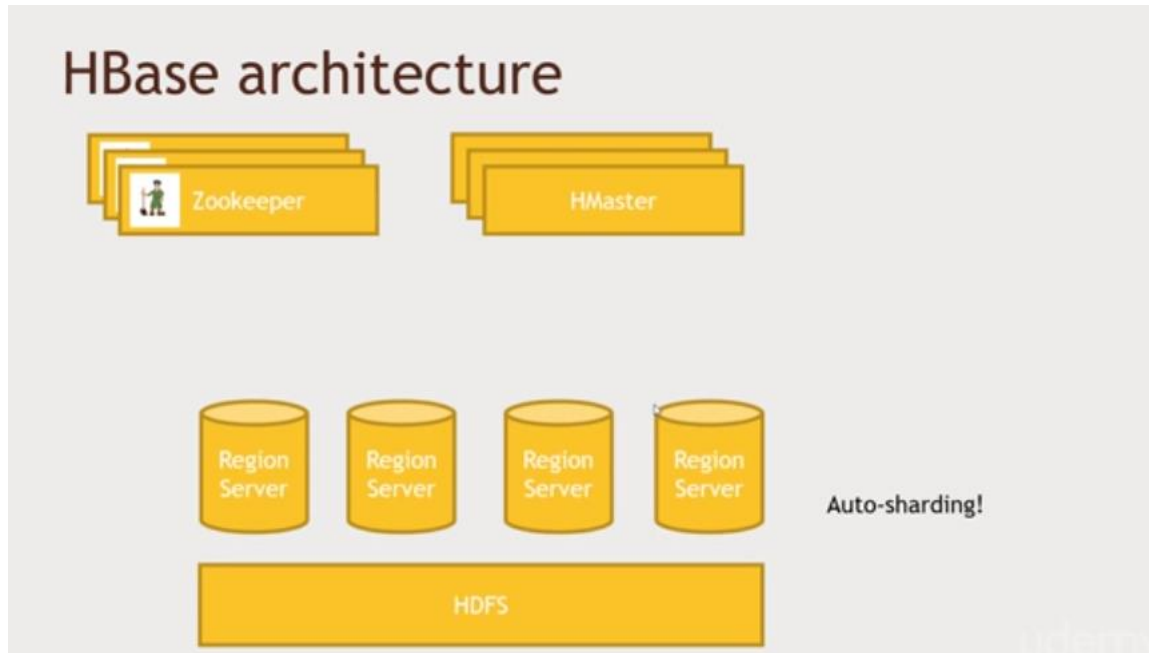
4. SECTION6 NOSQL

-HBASE-

- OLTP에서는 SQL이 사용하기 편리할 수 있다.
- LARGE SCALE, FAST면에서는 NOSQL이 적절하다.
- HBASE(NOSQL 시초). FAST SCALABLE TRANSACTION지원
EX)웹에 결과를 보내거나 그런 속도가 빠르며 LARGE DATA에 대해서 SCALABLE하다.
- 많은 DATA를 다룰 수 있는 것뿐 만 아니라 많은 CUSTOMER에 대한 REQUEST도 처리 가능하다. (REAL-TIME에 최적)
- KEY-VALUE 형태를 사용한다. EX)KEY에 대한 정보를 가져와라, KEY에 대한 정보를 넣어라.
- OLTP라도 SCALABLE이 필요가 없다면 MYSQL이 최적이다.



- WEB SERVER에서 SPARK STREAMING이나 FLUME통해서 HDFS에 정보 저장하고 MONGODB 통해서 QUERY 같이 접근한다.
- HBASE:
NON-RELATIONAL, SCALABLE, API를 가지고 CRUD OPEARATION지원 (CREATE, READ, UPDATE, DELETE)



- ➔ 여기서의 REGION SERVER란 KEY들의 RANGE등으로 SPLIT되어 있는 것으로 SHARDING(분산 저장) 혹은 RANGE PARTITIONING과 같다. DATA가 증가하더라도 자동으로 SHARDING, PARTITIONING과 같은 기능을 지원한다.

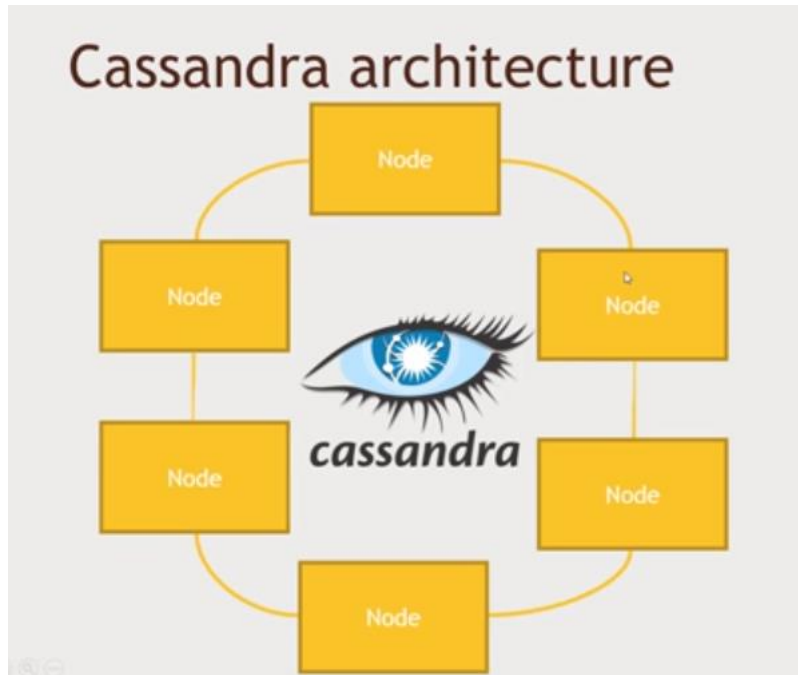
외부에서 요청이 온다면 REGION SERVER에 먼저 요청을 한다. HMASTER에서는 스키마 같은 것을 관리한다. ZOOKEEPER는 HMASTER가 DOWN되거나 이럴 때 알려준다. (FAULT-TOLERANCE)

COLUMN FAMILY하다. (DATA STORES SPARSELY IN COLUMNAL)

CELL을 가지고 TIMESTAMP를 가지고 여러 VERSION을 가진다.

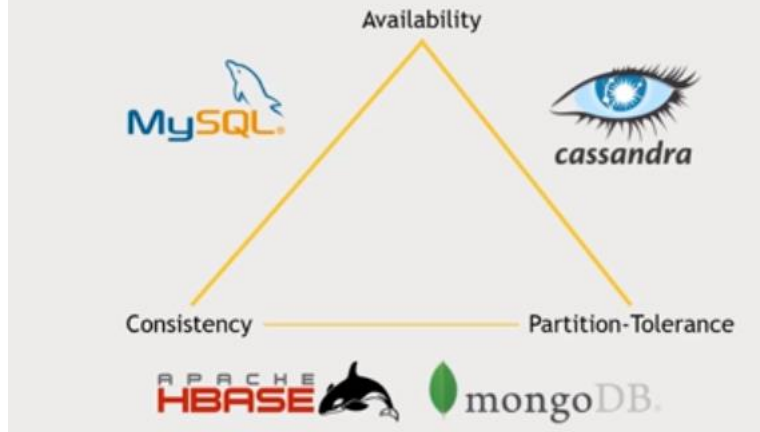
-CASSANDRA-

- ➔ MASTER NODE가 없어서 SINGLE POINT FAILURE가 없다
- ➔ CQL LANGUAGE지원
- ➔ CAP 3가지 중 2개 가질 수 있다.
(CONSISTENCY, AVAILABILITY, PARTITION-TOLERANCE).
- ➔ CASSANDRA의 경우는 AVAILABILITY와 PARTITION을 지원한다.
하지만 CUSTOMER가 임의로 ANSWER에 대해서 여러 NODE들의 값을 비교해서 CONSISTENCY가 많이 유지되지 않은 경우는 FAIL시킬 수 있다.



- ➔ HBASE와 달리 MASTER NODE가(NODE당 어떤 것을 SERVE 하는지 등에 대한 정보) 없다. 대신 GOSSIP PROTOCOL 형식으로 모든 NODE간의 COMMUNICATE를 한다. 그래서 CLIENT는 어떤 NODE에든 접근해서 진행하면 된다. VALUE RANGE에 따라서 HASH FUNCTION따라 할당된다. '이 때도 값에 대해서 3개 NODE를 봐서 2개 이상이 같은 값일 때만 받겠다.'란 식으로 CONSISTENCY를 유지할 수 있다.
- ➔ CQL을 지원하지만 SQL과는 달리 JOIN안된다. DATA는 DEMORNALIZED되어 있어야 한다.
- ➔ SPARK를 통해서 CASSANDRA의 DATA에 대해서 처리를 할 수 있다.

Where Cassandra Fits in CAP tradeoffs



-MONGODB-

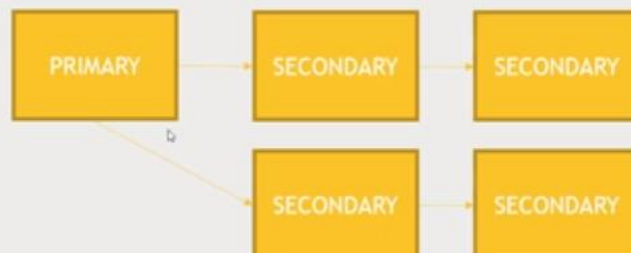
- DOCUMENT DATA MODEL을 다룬다. (JSON) <- FLEXIBLE.
- SINGLE MASTR를 가져서 CONSISTENCY를 보장한다.
- NO-SCHEMA다.

MongoDB terminology

- Databases
- Collections
- Documents

Replication Sets

- Single-master!
- Maintains backup copies of your database instance
 - *Secondaries can elect a new primary within seconds if your primary goes down*
 - *But make sure your operation log is long enough to give you time to recover the primary when it comes back...*



→ 그래서 BACKUP이 중요하다

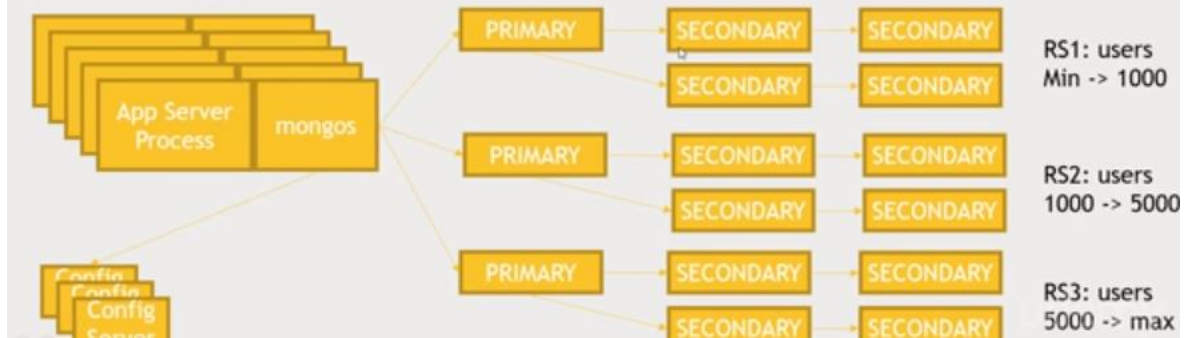
Replica Set Quirks

- A majority of the servers in your set must agree on the primary
 - *Even numbers of servers (like 2) don't work well*
- Don't want to spend money on 3 servers? You can set up an 'arbiter' node
 - *But only one*
- Apps must know about enough servers in the replica set to be able to reach one to learn who's primary
- Replicas only address durability, not your ability to scale
 - *Well, unless you can take advantage of reading from secondaries - which generally isn't recommended*
 - *And your DB will still go into read-only mode for a bit while a new primary is elected*
- Delayed secondaries can be set up as insurance against people doing dumb things

Sharding



- Finally - “big data”
- Ranges of some indexed value you specify are assigned to different replica sets



- ➔ APPLICATION은 먼저 3개 CONFIG SERVER에 접근해서 어떻게 PARTITIONED되어 있는지 혹은 REPLICATION이 어떤 식으로 되어있는지에 대한 정보를 얻는다. 그리고 PRIMARY에 접근하고 작업한다. SECONDARY는 PRIMARY에 대한 REPLICATION과 같은 것들이 있다. 이 때 이것들은 RANGE기준으로 나뉘어져 있다.

Sharding Quirks

- Auto-sharding sometimes doesn't work
 - Split storms, mongos processes restarted too often
- You must have 3 config servers
 - And if any one goes down, your DB is down
 - This is on top of the single-master design of replica sets
- MongoDB's loose document model can be at odds with effective sharding

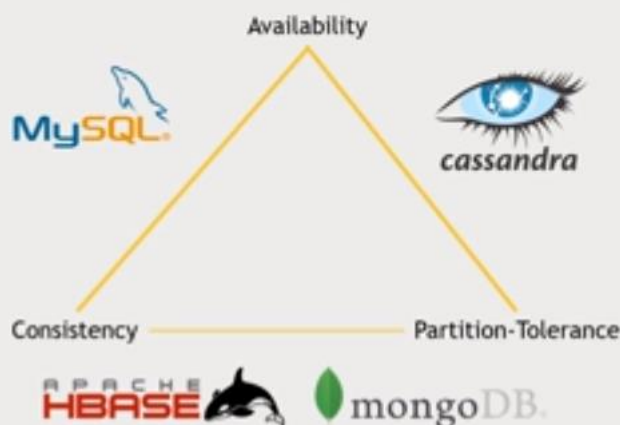
Neat Things About MongoDB

- It's not just a NoSQL database - very flexible document model
- Shell is a full JavaScript interpreter
- Supports many indices
 - *But only one can be used for sharding*
 - *More than 2-3 are still discouraged*
 - *Full-text indices for text searches*
 - *Spatial indices*
- Built-in aggregation capabilities, MapReduce, GridFS
 - *For some applications you might not need Hadoop at all*
 - *But MongoDB still integrates with Hadoop, Spark, and most languages*
- A SQL connector is available
 - *But MongoDB still isn't designed for joins and normalized data really.*

-CHOOSING DATABASE-

- ➔ SCALING REQUIREMENTS.
- ➔ SUPPORT CONSIDERATIONS. SECURITY를 생각한다면 MONGODB가 좋다.
- ➔ BUDGET. 하지만 AWS같은데서 해결 가능하다.
- ➔ CAP CONSIDERATIONS. 여기서 PARTITION-TOLERANCE는 기본적으로 지켜줘야 하고 AVAILABILITY나 CONSISTENCY중에서 선택해야 한다.

CAP considerations



→ SIMPLICITY. 필요할 경우가 아니면 SIMPLE하게 해라. 제일 중요하다.

- You're building an internal phone directory app
 - *Scale: limited*
 - *Consistency: Eventual is fine*
 - *Availability requirements: not mission critical*
 - *MySQL is probably already installed on your web server...*

→ MYSQL 쓰는 것이 제일 좋다.

Another example

- You want to mine web server logs for interesting patterns
- What are the most popular times of day? What's the average session length? Etc.

→ IMPORT HDFS해서 분석하면 된다.

Another example

- You have a big Spark job that produces movie recommendations for end users nightly
- Something needs to vend this data to your web applications
- You work for some huge company with massive scale
- Downtime is not tolerated
- Must be fast
- Eventual consistency OK - it's just reads

→ NOSQL 中 CASSANDRA. CONSISTENCY가 별로 중요하지 않는 경우라서 CASSANDRA 선택.