실용주의 하둡(Pragmatic Hadoop)

2013년 4월

[socurites@gmail.com](mailto:socurites@gmail.com)

**책의 컨셉**

웹서버에서 실시간으로 쏟아내고 있는 대량의 웹로그를 관리하는 시스템을 구축한다. 로그 관리 시스템을 구축하기 위해서는 데이터를 수집 > 저장 > 정제 > 분석 및 검색 > 관리하는 단계를 거쳐야 한다.

이 책에서는 각 단계별로 필요한 하둡 기반의 오픈소스를 빠르게 적용하여 구축하는 사례를 중심으로 설명한다. 1부 “적용하기”에서는 기본 개념을 상세히 설명하는 대신, 시스템을 구축할 수 있을 정도로만 기본 개념만 간단히 설명한다.

각 오픈소스별로 상세한 내용은 2부 “이해하기”에서 다룬다. 설치 및 설정하는 방법뿐만 아니라, 주의사항 등을 포괄적으로 설명하여 오픈소스를 깊이 이해할 수 있도록 한다.

3부 “운영하기”에서는 장애에 대응하기 위한 방법과 성능향상을 위한 팁 등 고급 개념을 설명한다.

1부를 통해 기본 개념을 이해하고 실제로 적용해봐야만 하둡을 더욱 쉽게 이해할 수 있다고 본다. 그래서 상세한 개념을 먼저 설명하는 다른 책과는 달리 이 책은 먼저 적용을 한 후, 관련된 내용은 필요에 따라 찾아볼 수 있도록 구성했다.

**대상 독자**

기본적으로 이 책은 대량의 데이터를 수집하여 분석하려는 사람이라면 누구나 쉽게 이해할 수 있도록 작성했다. 하둡을 접해본 적이 없는 독자라도 쉽게 접근할 수 있도록 쓰려고 노력했다. 웹로그 분석 시스템을 구축하는 사례를 중심으로 설명하지만, 기본 개념은 범용적으로 활용할 수 있다.

**목차**

[1부 적용하기 6](#_Toc355355648)

[1. 도입 예제 7](#_Toc355355649)

[ 데이터 이동 흐름 7](#_Toc355355650)

[ 지금까지의 아키텍처 8](#_Toc355355651)

[2. 수집하기 9](#_Toc355355652)

[ 설치하기 9](#_Toc355355653)

[3. 저장하기 17](#_Toc355355654)

[ 하둡 파일 시스템 설치하기 17](#_Toc355355655)

[ 환경 설정하기 18](#_Toc355355656)

[ SSH 설정하기 20](#_Toc355355657)

[ 네임 노드 구동하기 20](#_Toc355355658)

[ 데이터 노드 구동하기 21](#_Toc355355659)

[ 플럼과 연동하기 21](#_Toc355355660)

[4. 정제하기 22](#_Toc355355661)

[ 환경 설정하기 22](#_Toc355355662)

[ 잡 트래커 구동하기 23](#_Toc355355663)

[ 태스크 트래커 구동하기 23](#_Toc355355664)

[ 맵리듀스 프로그래밍하기 23](#_Toc355355665)

[ 하이브 사용하기 23](#_Toc355355666)

[5. 분석하기 26](#_Toc355355667)

[6. 관리하기 30](#_Toc355355668)

[7. 검색하기 30](#_Toc355355669)

[2부 이해하기 31](#_Toc355355670)

[8. 플럼(Flume) 32](#_Toc355355671)

[ 개요 32](#_Toc355355672)

[ 설치 37](#_Toc355355673)

[ 의사 분산 모드로 설치하기 38](#_Toc355355674)

[ 완전 분산 모드로 Flume 설치하기 38](#_Toc355355675)

[ 고급 기능 사용하기 38](#_Toc355355676)

[9. 하둡 분산 파일 시스템(HDFS) 41](#_Toc355355677)

[ 하둡 파일 시스템 구성 41](#_Toc355355678)

[ 설치 44](#_Toc355355679)

[ 실행 47](#_Toc355355680)

[ 하둡 파일 시스템 탐색 49](#_Toc355355681)

[ 파일 시스템 API 활용 50](#_Toc355355682)

[10. 맵리듀스(MapReduce) 51](#_Toc355355683)

[ 맵리듀스 시스템 구성 51](#_Toc355355684)

[ 맵리듀스 프로그램 53](#_Toc355355685)

[ 설치 56](#_Toc355355686)

[11. 맵리듀스 프로그래밍 59](#_Toc355355687)

[ WordCount 예제 59](#_Toc355355688)

[ 맵리듀스 프로그래밍 구성 요소 60](#_Toc355355689)

[ 맵리듀스 프로그래밍 개발 절차 62](#_Toc355355690)

[ 공통 데이터 배포 67](#_Toc355355691)

[ 정렬 70](#_Toc355355692)

[ 조인 70](#_Toc355355693)

[12. 하이브(Hive) 71](#_Toc355355694)

[13. 피그(Pig) 79](#_Toc355355695)

[14. HBase 80](#_Toc355355696)

[ NoSQL 개요 80](#_Toc355355697)

[ HBase 소개와 구조 82](#_Toc355355698)

[ 설치 83](#_Toc355355699)

[ 실행 84](#_Toc355355700)

[15. 캐스캐이딩(Cascading) 85](#_Toc355355701)

[16. 우지(oozie) 91](#_Toc355355702)

[17. elasticsearch 92](#_Toc355355703)

[3부 운영하기 98](#_Toc355355704)

[18. 하둡 운영하기 98](#_Toc355355705)

[ 실행과 관리 98](#_Toc355355706)

[ 네임노드 백업 및 HA 구성 99](#_Toc355355707)

[19. 맵리듀스 성능향상하기 103](#_Toc355355708)

[ 맵 태스크 튜닝 103](#_Toc355355709)

[ 리듀스 태스크 튜닝 103](#_Toc355355710)

[ 병합 정렬 튜닝 104](#_Toc355355711)

[ 잡트래커/태스크 트래커 튜닝 104](#_Toc355355712)

[20. 맵리듀스 프로그래밍 활용 팁 106](#_Toc355355713)

[21. 하이브 성능향상하기 109](#_Toc355355714)

[22. 검색 인덱스 관라하기 120](#_Toc355355715)

[부록 121](#_Toc355355716)

[23. 부록 1 웹로그 포맷 121](#_Toc355355717)

[24. 부록 2 polliwog를 사용한 웹로그 분석 132](#_Toc355355718)

[25. 부록 3. Awstats를 사용한 웹로그 분석 134](#_Toc355355719)

[26. 부록 4. [번역] 템플톤(Templeton)이란 135](#_Toc355355720)

[27. 부록 5. ANTLR을 사용한 웹로그 파서 만들기 138](#_Toc355355721)

[28. 부록 6. 하둡 분산 파일 시스템을 기반으로 색인하고 검색하기 145](#_Toc355355722)

[29. 부록 7. Splunk 154](#_Toc355355723)

[30. 부록 8. 플럼 OG vs. 플럼 NG 165](#_Toc355355724)

[31. 부록 9. 샘플 웹로그 166](#_Toc355355725)

[32. 부록 10. 테스트 주도로 하둡 맵리듀스 프로그래밍하기 167](#_Toc355355726)

[33. 부록 11. 하둡 배포판 178](#_Toc355355727)

[ 클라우데라 CDH 178](#_Toc355355728)

[ 호튼웍스 HDP 178](#_Toc355355729)

[ MapR 테크놀로지스 MapR Editions 178](#_Toc355355730)

[ EMC GreenPlum UAP 178](#_Toc355355731)

[ 아마존 EMR 178](#_Toc355355732)

[ HStreaming 178](#_Toc355355733)

[참고자료 179](#_Toc355355734)

# 1부 적용하기

## 도입 예제

당신의 회사나 고객사에서 수백 또는 수천 대의 웹 서버를 운영하는 상황을 생각해보자. 그리고 웹 서버에서 수십MB에서부터 수GB에 이르는 웹 로그를 매일 생성한다고 해보자. 당신은 웹 사이트 방문자가 웹사이트를 어떻게 이용하는지, 그리고 해커가 웹서버를 대상으로 악의적인 행위를 하지 않는지 분석하기 위해 수백TB에 달하는 웹 로그를 저장해 왔다. 인프라에 대한 투자가 넉넉한 회사였다면 NAS나 SAN을 구축하여 웹 로그를 저장하는데 어려움이 없었을 수도 있다. 하지만 대개의 경우 대량의 웹 로그를 저장하기 위해, 여기저기 흩어져 있는 파일서버에서 빈 공간을 찾느라 헤매고 다녔을 게 뻔하다.

뿐만 아니라 고객 프로모션을 한달 동안 진행한 후 방문자가 웹 사이트에 사용하는 방식에 변화가 있었는지 분석하거나, 보안사고가 발생하여 해킹 흔적을 찾기 위해 몇 일 밤낮을, 심지어는 몇 주에 걸쳐서 한 달치 웹 로그를 뒤진 경험이 있을 수도 있다.

하둡(Hadoop)의 등장으로 이처럼 혼란스러울뿐더러 지겨운 업무에서 이제는 해방될 수 있다. 값싼 리눅스 장비를, 심지어는 오래되어 쓰지 않는 PC급 장비를 사용해서 안정적인 데이터 저장소를 구축할 수 있다. 게다가 맵리듀스(MapReduce) 프로그램을 이용하면, 한 달치 데이터를 분석하는 일도 수분 정도로 단축할 수 있다.

이제 당신은 하둡 에코시스템을 기반으로 대량의 웹 로그를 저장하여 분석하는 시스템을 구축하여 여러분의 동료를 빅데이터의 악모으로부터 해방시키려고 한다.

먼저 데이터가 생성되는 출발점인 웹 서버에서, 데이터가 저장되는 하둡 파일 시스템에 이르기까지 데이터가 이동하는 흐름부터 살펴보자.

### 데이터 이동 흐름

<그림. 데이터 이동 흐름>

데이터 생성 서버 > 데이터 수집 서버 > 데이터 저장 서버

웹 서버 > 수집 서버 > 저장 서버

방문자가 웹 사이트에 접속하면 웹 서버는 관련된 정보를 로그 파일에 실시간으로 기록한다. 일반적으로 로그 파일은 하루 단위로 롤링되며, 아래와 같이 날짜 단위로 관리한다.

<그림. 저장된 로그 파일>

access.log

access.log\_20130401

access.log\_20130402

다수의 웹 서버에 저장되어 있는 웹 로그를 저장 서버, 즉 하둡 파일 시스템에 저장하기 위해서는, 분산되어 있는 웹 서버로부터 로그 파일을 수집해야 한다. 수집 서버가 이 역할을 담당한다. 즉 수집 서버는 웹 서버에 접근하여 실시간 또는 특정 주기로 로그 데이터를 가져 온 후, 저장 서버에 저장하는 역할을 담당한다.

### 지금까지의 아키텍처

이처럼 데이터를 수집하여 저장하기 위해서는 아래와 같이 아키텍처 레이어를 구성해야 한다.

<그림. 수집 저장을 위한 아키텍처>

데이터 생성 서버 > 데이터 수집 서버 > 데이터 저장 서버

웹 서버 > 수집 서버 > 저장 서버

수집 에이전트 > 수집 데몬 > 하둡 파일 시스템

수집 에이전트는 웹 서버의 로그를 실시간으로 수집 서버로 전송한다. 수집 데몬은 특정 포트로 대기하고 있다가 수집 에이전트가 전송한 웹 로그를 수신한 후, 하둡 파일 시스템에 저장하게 된다.

먼저 웹 로그를 수집하는 부분, 즉 수집 에이전트와 수집 데몬부터 살펴보자.

## 수집하기

분산되어 위치한 웹 서버로부터 데이터를 수집하는 방식에는 수집 방향에 따라 푸시와 폴링, 두 가지 방법이 있다. 푸시는 데이터를 생성한 장비에서 수집 서버로 보내는 방식이며, 반면 폴링은 수집 서버에서 데이터 생성 장비에 접근하여 데이터를 직접 가져오는 방식이다.

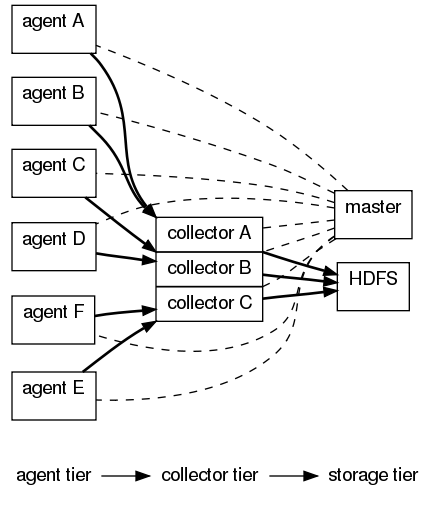
여기에서는 플럼(Flume)을 사용하여 웹 로그를 수집하려고 한다. Flume은 에이전트 기반의 수집 프레임워크로, 데이터 생성 서버에 에이전트 데몬을 설치하고 수집 서버에는 컬렉터 데몬을 설치한다. 에이전트는 웹로그를 컬렉터로 실시 전송하는 역할을 하며, 컬렉터는 전송받은 데이터를 저장 서버로 저장하는 역할을 담당한다. 그리고 컬렉터와 에이전트를 관리할 수 있는 마스터 데몬이 있다.

플럼은 푸시와 풀링 방식 모두를 지원할 뿐만 아니라, 에이전트를 설치하지 않더라도 syslog와 snmp 프로토콜을 통해 수신할 수 있는 agentless 방식도 지원한다.

## 설치하기

플럼을 설치하기 전에 먼저 수집할 대상 웹로그를 다운로드한다. 운영 환경이라면 웹서버에서 실시간으로 생성하는 로그를 수집해야 하지만 설명의 편의를 위해 이미 생성된 웹 로그 파일을 수집하도록 한다. 하지만 운영 환경에 적용하더라도 그 개념은 동일하니 어려움은 없으리라고 본다. 웹로그 샘플과 관련된 내용은 “부록 9. 샘플 웹로그”를 참조한다.

플럼은 그 목적에 따라 에이전트, 컬렉터, 마스터 노드로 분류할 수 있다. 에이전트는 로그를 생성하는 대상 장비에 설치하여 로그를 전송한느 역할을 한다. 컬렉터는 에이전트가 전송한 로그를 수신하여 저장소에 저장하는 역할을 한다. 마스터는 에이전트와 컬렉터를 관리하는 역할을 맡는다.



따라서 로그를 수집하기 위해서는 총 3종류의 노드, 즉 에이전트, 컬렉터, 마스터를 설치해야 한다. 결국 최소한 3개의 머신이 있어야만 플럼 클러스터룰 구성할 수 있다.

하지만 편의를 위해 플럼은 하나의 장비에 에이전트와 컬렉터, 마스터를 설치하여 테스트해 볼 수 있는 의사 분산 모드를 지원한다. 1부에서는 하나의 장비에서 테스트 환경을 구성하려고 한다.

플럼은 RPM, 데비안 패키지, tarball 등의 형식으로 제공된다. 여기에서는 tarball을 다운로드 받아 설치하고자 한다. 플럼은 자바 기반으로 개발되었으므로 테스트 서버에는 JRE 1.6 이상이 설치되어 있어야 한다.

클라우데라 GitHub(<https://github.com/cloudera/flume/downloads>)에서 플럼을 다운로드한다. 여기에서는 현재 최신 버전인 flume-distribution-0.9.4-bin.tar.gz을 사용한다. 설치는 간단한다. 단순히 테스트 서버의 적당한 위치에 압축을 풀기만 하면 된다. 편의를 위해, tarball이 풀린 위치를 flume으로 심볼릭 링크를 걸어둔다.

ln -s flume-distribution-0.9.4/ flume

시스템 환경 변수에 설정 디렉토리 정보를 FLUME\_CONF\_DIR이라는 이름으로 등록한다.

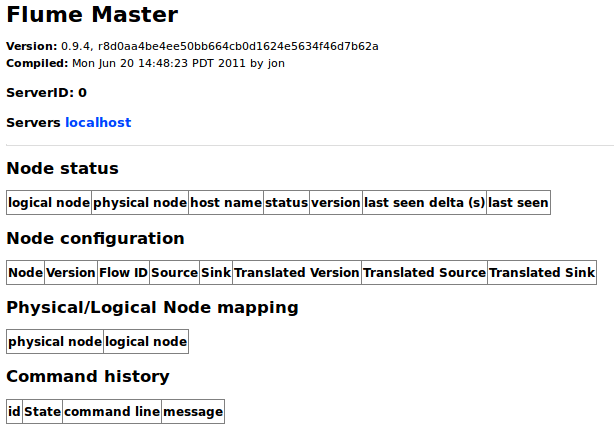
#FLUME  
export FLUME\_HOME=/home/sunny/PlayGrounds/flume/flume  
export FLUME\_CONF\_DIR=$FLUME\_HOME/conf

### 마스터 기동하기

먼저 마스터를 실행한다.

bin/flume master

마스터가 정상적으로 실행이 완료되면, 플럼 마스터 웹 UI에 접속할 수 있다. 마스터 웹 UI 접속 주소는 <http://localhost:35871> 다.



* 노드 상태(Node status)  
  노드(컬렉터 또는 에이전트)를 실행한 경우, 해당 노드의 상태 정보를 보여준다.
* 노드 설정(Node configuration)  
  노드에서 실행할 설정 정보를 보여준다. 아직 노드에 할당되지 않은 설정들만 보여준다.
* 물리/논리 노드 매핑(Physical/Logical Node mapping)  
  설정 정보와 실제 노드 사이에 매핑된 정보를 보여준다.
* 명령 히스토리(Command history)  
  마스터 웹 UI를 통해 실행한 명령(설정 생성, 설정 정보 매핑 등)의 성공 여부를 표시힌다.

아직 어떤 노드도 실행하지 않았으므로, 웹 UI의 정보는 비어 있는 상태임을 확인할 수 있다. 이제 컬렉터와 에이전트를 실행해 보자.

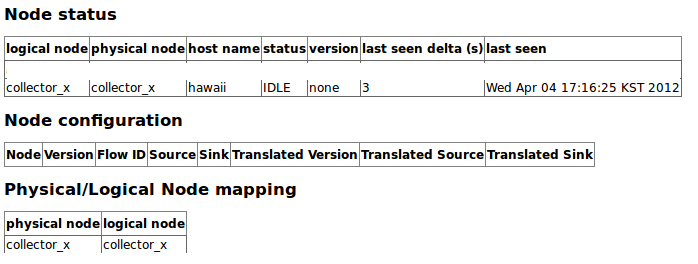
### 컬렉터 기동하기

먼저 컬렉터를 실행하자. 플럼이 설치된 실제 장비에서 아래와 같이 실행되는 노드는 모두 물리 노드(Physical Node)라고 부른다. 하나의 물리 노드에는 다수의 논리 노드(Logical Node)를 실행할 수 있다. 실제로 데이터를 수집하고 전송하는 일은 논리 노드에서 담당한다. 반면 물리 노드에서는 자신이 실행 중인 논리 노드에 대한 자원 등을 관리한다.

플럼 노드는 반드시 이름을 가지고 있어야 하며, -n 옵션으로 지정할 수 있다.

bin/flume node\_nowatch -n collector\_x

마스터 웹 UI에 접속하면 아래와 같이 컬렉터가 등록되었음을 확인할 수 있다.



뿐만 아니라 물리 노드의 이름(colloctor\_x)과 동일한 이름을 가지는 논리 노드가 실행되었음을 노드 상태에서 확인할 수 있다. 기본적으로 물리 노드가 실행되면 자동으로 논리 노드 하나가 실행이 되는데, 이러한 논리 노드를 기본 논리 노드(Primary Logical Node)라고 부른다. 또한 status가 IDLE 상태인데,

**[[status 관련 내용 추가할 것]]**

### 에이전트 기동하기

이제 에이전트를 실행하자.

bin/flume node\_nowatch -n agent\_a

에이전트를 기동하는 명령은 컬렉터를 기동하는 명령과 동일하다. 사실 플럼에는 에이전트와 컬렉터 사이의 구분이 전혀 없다. 단지 역할에 따라 논리적으로 레이어를 구분할 뿐이다. 간단히 말하면 맨 앞단에서 데이터를 수집하여 전송하는 티어를 에이전트 티어라고 부르며, 에이전트가 전송한 데이터를 수신하는 티어를 컬렉터라고 부른다.

컬렉터용 물리 노드와 에이전트용 물리 노드가 정상적으로 실행이 되었다면, 이제 논리 노드를 각각 실행하여 로그 데이터를 수지해야 한다.

### 설정하기

[[ 작성중 ]]

에이전트는 콘솔로부터 데이터를 입력받아, 컬렉터로 전송한다.   
컬렉터는 에이전트로부터 전송받은 로그를 다시 콘솔로 출력한다.

|  |
| --- |
| http://www.gurubee.net/download/attachments/26739848/flume_test_architecture.png |

노드에 대한 설정은 Flume Master 웹 화면의 "config" 메뉴에서 실행한다.   
"Configure multiple nodes" 밑의 텍스트 영역에 아래의 명령어를 입력한 후, "Submit query' 버튼을 누른다.  
agent\_a : console | agentSink("localhost", 35853);  
collector\_x : collectorSource(35853) | console;

|  |
| --- |
| http://www.gurubee.net/download/attachments/26739848/flume_mast_web_test_config.png |

Flume Master 웹 화면의 메인으로 돌아오면 아래와 같이, Command history 영역에서 명령어가 성공했음을 알 수 있다.   
또한, Node configuration에 보면 source와 sink가 제대로 설정되었음을 확인할 수 있다.   
그리고 Node status 영역에서는 각 노드의 상태가 "ACTIVE" 상태로 바뀌었음을 볼 수 있다.

|  |
| --- |
| http://www.gurubee.net/download/attachments/26739848/flume_master_web_pseudo_config.png |

로그 생성 및 수집하기

에이전트 노드를 실행한 콘솔로 가서, 아무런 텍스트를 입력한 후 엔터키를 누른다.  
그리고 나서 컬렉터 노드를 실행한 콘솔로 가 보면, 에이전트에서 전송한 로그가 콘솔에 출력됨을 확인할 수 있다.

결론

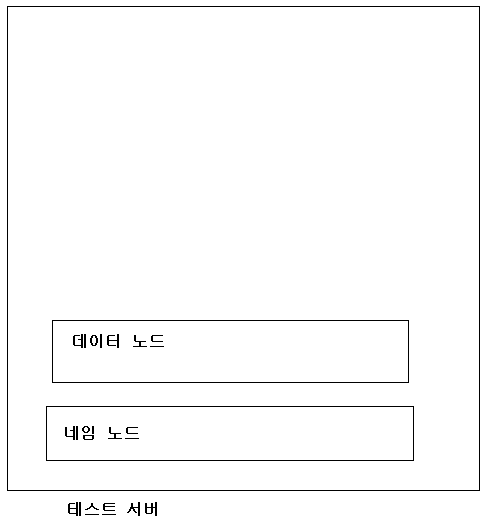
여기에서는 단일 노드에서 마스터 / 컬렉터 /에이전트 노드를 모두 실행했다.   
실제 환경이라면 마스터는 별도의 마스터 서버에서, 컬렉터는 로그를 수집할 서버에, 에이전트는 로그를 생성할 서버(웹서버 등)에 설치해서 실행할 것이다.  
여기에서는 기본 논리 노드에 sink | source 정보를 바로 설정했다.   
이와는 달리, 설정 정보만을 담고 있는 별도의 논리 노드를 만든 후, 해당 논리 노드를 물리 노드에 매핑(map 명령어를 이용해서)하는 방식도 있다.

|  |
| --- |
| http://www.gurubee.net/download/attachments/26739848/flume_master_web_raw_command.png |

## 저장하기

이제 로그 데이터는 웹 서버로부터 수집 서버로 전송을 완료했다. 전송 서버가 수신한 데이터는 영속성이 있는 데이터 저장소, 즉 하둡 파일 시스템에 저장해야 한다. 하둡 파일 시스템은 네임 노드와 데이터 노드로 구성된다. 실제 데이터는 데이터 노드에 저장되며, 네임 노드는 데이터 노드를 관리하는 역할을 한다.

하둡은 독립 실행 모드(Standalone Mode), 의사 분산 모드(Pseudo-Distributed Mode), 완전 분산 모드(Fully Distributed Mode)로 실행할 수 있다. 하둡의 실행 모드에 대한 내용은 “하둡 실행 모드”를 살펴보라. 여기에서는 의사 분산 모드로 하둡 파일 시스템을 구성하려고 한다. 의사 분산 모드로 하둡을 구성하면 다음과 같다.



### 하둡 파일 시스템 설치하기

먼저 아파치 하둡 릴리즈 페이지에서 안정화된 배포판을 다운로드 받는다. 여기에서는 hadoop-0.20.2.tar.gz를 다운로드 받는다.

로컬 파일 시스템의 적당한 위치에 압축을 푼다.

$ tar xzf hadoop-0.20.2.tar.gz

이렇게 하둡이 설치된 디렉토리를 HADOOP\_INSTALL이라고 부르자.

다음으로, HADOOP\_INSTALL/conf/hadoop-env.sh 파일을 열어, JAVA\_HOME 변수를 아래와 같이 설정한다.

export JAVA\_HOME=/usr/lib/jvm/java-6-sun

필수사항은 아니지만 하둡 관련 명령어를 쉽게 실행하기 위해 하둡 설치 디렉토리(HADOOP\_INSTALL)를 환경 변수에 설정하고, PATH를 설정한다. 여기에서는 .bashrc 파일에 설정했다.

#HADOOP

export HADOOP\_INSTALL=하둡이 설치된 경로

export PATH=$PATH:$HADOOP\_INSTALL/bin

설정한 환경변수를 반영한다.

$ source ~/.bashrc

아래와 같이 하둡 명령어를 실행해서 제대로 설치죄었는지 확인해 볼 수 있다.

$ hadoop version

Hadoop 0.20.2

Subversion https://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20 -r 911707

Compiled by chrisdo on Fri Feb 19 08:07:34 UTC 2010

### 환경 설정하기

하둡 파일 시스템 환경은, 실행 모드에 따라 달리 설정해야 한다.

#### 실행 모드

하둡은 목적에 따라 3가지 실행 모드를 지원한다.

* 독립 실행(Standalone) 모드  
  하둡을 설치하면 기본적으로 독립 실행 모드로 설정된다. 이 모드를 사용하면, 분산 파일 시스템이 아닌 로컬 파일 시스템을 사용하도록 구동된다. 따라서 분산 환경에서 발생할 수 있는 문제가 일어나지 않으므로, 맵리듀스 프로그램을 빠르게 개발하고 디버깅하기에 편리하다.
* 의사 분산(Pseudo-distributed) 모드  
  하둡 관련 데몬(네임노드, 데이터 노드, 잡 트래커, 태스크 트래커)을 한 장비에서 모두 구동하여, 마치 분산 환경처럼 실행하도록 설정한다. 따라서 실제 분산 환경과 동일한 환경으로 구성되기 떄문에, 완전 분산 모드로 운영중인 서버에 대한 테스트 환경으로 사용할 수 있다.
* 완전 분산(Fully distributed) 모드  
  여러 대의 장비로 구성된 경우 클러스터로 운영할 모드

사용하려는 실행 모드에 따라 하둡 환경을 설정해야 한다. 1부에서는 의사 분산 모드로 설정하고,완전 분산 모드로 설정하는 방법은 2부에서 설명하도록 하겠다.

하둡은 XML 파일을 사용해서 설정하며, 설정할 파일은 총 3가지다.

* conf/core-site.xml

하둡 컴포넌트의 공통적인 속성을 정의한다.

* conf/hdfs-site.xml

하둡 파일 시스템과 관련된 설정을 한다.

* conf/mapred-site.xml

하둡 맵리듀스와 관련된 설정을 한다.

먼저 core-site.xml부터 설정하자.

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>

<name>**fs.default.name**</name>

<value>hdfs://localhost:9000</value>

</property>

</configuration>

fs.default.name에 하둡 파일 시스템의 네임노드의 호스트와 포트를 설정한다. 하둡 파일 시스템의 기본 포트는 8020이며, 여기에서는 9000을 포트로 설정했다.

다음으로 hdfs-site.xml을 설정한다.

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>

<name>**dfs.replication**</name>

<value>1</value>

</property>

</configuration>

파일을 몇 개까지 복사해서 저장할 것인지 복제 계수를 지정한다. 일반적으로 완전 분산 모드에서는 3개의 복사본을 유지하도록 설정한다. 하지만 의사 분산 모드에서는 반드시 1로 설정하여야 한다.

mapred-site.xml은 분산 처리를 위한 맵리듀스를 위한 설정이며, 4장 정제하기에서 설정하도록 하겠다.

### SSH 설정하기

의사 분산 모드/완전 분산 모드에서는 모두 데몬 프로세스를 실행하므로, 원격 서버의 명령을 수행할 수 있도록 ssh를 설정해야 한다. ssh가 설치되지 않은 경우 먼저 ssh를 설치하도록 한다. ssh 명령어가 실행가능하더라도 ssh 데몬이 실행되어 있지 않으면 SSH을 설정할 수 없으므로 아래와 같이 ssh를 다운받아서 설치한다.

$sudo apt-get install ssh

이제 아래와 같이 ssh 키를 생성한다.

$ ssh-keygen -t rsa -P '' -f ~/.ssh/id\_rsa

$ cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys

아래와 같은 명령어를 실행행서 성공한다면 제대로 설정한 것이다.

$ ssh localhost

Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)

\* Documentation: https://help.ubuntu.com/

Last login: Fri Nov 11 17:25:48 2011 from localhost

아래와 같이 22번 포트에 접속할 수 없다는 에러메시지가 나는 경우

ssh: connect to host localhost port 22: Connection refused

$ pgrep sshd

를 실행한다. 만약 아무런 결과도 나오지 않는다면, sshd이 설치되지 않았거나, 실행중이 아닌 경우다.

### 네임 노드 구동하기

하둡 파일 시스템을 사용하려면 먼저 네임 노드를 포맷해야 한다.

$ hadoop namenode -format

하둡 파일 시스템과 맵리듀스 데몬을 아래와 같이 구동한다.

$ bin/hadoop-deamon.sh namenode start

### 데이터 노드 구동하기

데이터 노드를 실행한다.

$ bin/hadoop-deamon.sh datanode start

이 제대로 실행되었는지 확인하는 방법에는 여러가지가 있다. 그 중에서도 웹 UI를 통해 확인할 수도 있다. 네임노드는 <http://localhost:50070/>에서 확인할 수 있다.

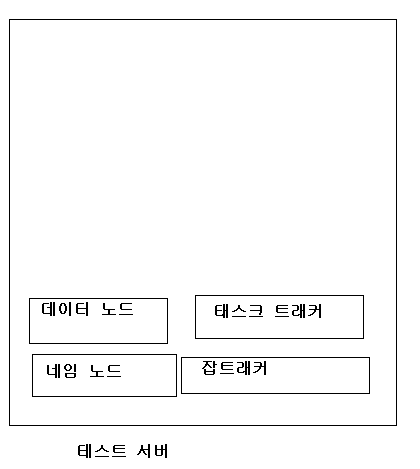
[[ 그림. 네임노드 웹 UI ]]

### 플럼과 연동하기

[[ 플럼에 대해 HadoopSink 만들 것 ]]

## 정제하기

하둡 파일 시스템에 저장된 웹 로그로에서 클라이언트 IP가 1.1.1.1에 해당하는 기록을 모두 찾아야 한다고 해보자. 기본적으로 하둡 파일 시스템의 데이터를 분석하기 위해서는 맵리듀스 프레임워크를 활용해야 한다. 맵리듀스는 잡 트래커와 태스크 트래커로 구성된다.



### 환경 설정하기

3장 저장하기에서 설정하지 않았던 mapred-site.xml을 마지막으로 설정한다.

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>

<name>mapred.job.tracker</name>

<value>localhost:8021</value>

</property>

</configuration>

### 잡 트래커 구동하기

$ bin/hadoop-deamon.sh jobtracker start

### 태스크 트래커 구동하기

$ bin/hadoop-deamon.sh tasktracker start

네임노드와 마찬가지로 잡트래커도 웹 UI.에서 확인할 수 있으며 접속 주소는 <http://localhost;50030/> 이다.

### 맵리듀스 프로그래밍하기

이제 클라이언트 IP가 1.1.1.1인 경우를 포함하는 웹 로그를 찾아보자.

#### 워드 카운트 예제

[[ 워드 카운트 예제 ]]

테스트 주도 개발(TDD)에 관심이 있다면, 테스트 주도로 맵리듀스 하는 프로그래밍과 관련된 내용은 부록을 참고하기 바란다.

앞서 본 맵리듀스 프로그램에서는 매번 로그 라인을 분할한 후, 분할된 데이터를 사용했다. 대개의 경우 하나의 라인보다는 특정 필드 단위로 로그를 분석해야 하는 경우가 많다. 따라서 라인 단위가 아니라 필드 단위로 로그를 분할하여 저장하는 편이 재사용 측면에서 효율적이다.

이러한 역할을 할 수 있는 프레임워크가 하이브다. 하이브를 사용하면 간단한 처리를 위해 매번 맵리듀스 프로그래밍을 하지 않아도 된다. 다음에는 하이브에 대해서 알아보자.

### 하이브 사용하기

하이브는 마치 RDBMS와 유사하게 컬럼 단위로 데이터를 관리할 수 있다. 뿐만 아니라 앞의 간단한 유스케이스를 위해서 맵리듀스 프로그램을 만드는 일은 너무나 번거로운 일이다. 하이브를 사용한다면 앞서 만든 맵리듀스 프로그램을 아래와 같이 간단한 SQL문을 사용하여 해결할 수 있다..

<그림. SQL문을 사용하여 데이터 조회>

select \* from weblog where client\_ip = ‘1.1.1.1’

이제 하이브를 사용하여 사용자 IP가 1.1.1.1인 사용자를 찾아보도록 하자.

#### 하이브 설치하기

하이브를 다운로드 한 후 압축을 적당한 위치에 푼다. 하이브를 실행하려면 HADOOP\_HOME을 환경변수로 추가해야 한다. $HIVE\_HOME/conf/hive-env.sh 파일에 HADOOP\_HOME 변수를 설정한다. 만약 hive-env.sh 파일이 없다면, hive-env.sh.template을 복사하여 만든다.

$ cp conf/hive-env.sh.template conf/hive-env.sh

$ vi conf/hive-env.sh

그리고 필요한 부분을 주석을 제거한 후, 자신의 환경에 맞게 설정한다.

# Set HADOOP\_HOME to point to a specific hadoop install directory

HADOOP\_HOME=/Users/socurites/Works/Hadoop/hadoop

이렇게 하이브 설치가 끝나면, 이제 하이브 콘솔로 접속할 수 있따.

$ bin/hive

간단히 테이블 목록을 조회해 보자.

hive> show tables

설치 후 명령어를 처음 실행하는 경우, 로컬 파일 시스템에 하이브 메타 정보를 저장할 메타스토어를 구성하게 된다. 하이브 콘솔을 접속한 위치에 아래와 같이 메타 스토어 디렉토리가 생성된 것을 확인할 수 있다.

[[ 그림. 로컬 메타스토어에 생성된 디렉토리 ]]

기본적으로 하이브는 메타 정보를 로컬 파일 시스템에 기록한다. 하지만 여러 사용자가 동시에 사용해야 한다면 MySQL을 메타 스토어로 사용하는 로컬 모드로 설정해야 한다. 이외에도 원격 접속을 허용하는 원격 서비스 모드도 있다. 자세한 내용은 11장 하이브를 참고하라.

#### 테이블 생성하기

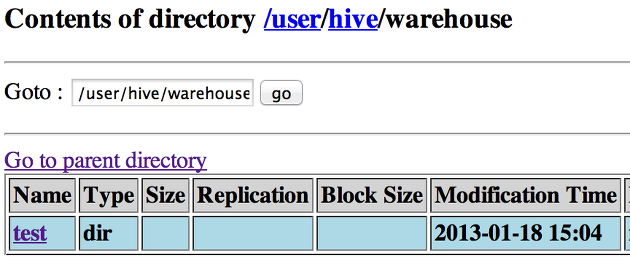
TBD

**명령어 실행하기**

테이블을 하나 만들어 보자

*hive> create table (id string) ;*

테이블이 정상적으로 생성되면 HDFS에 아래와 같은 경로에 테이블 이름과 동일한 디렉토리가 만들어진다.



하이브는 /user/hive/warehouse 를 테이블의 저장공간으로 최상위 공간으로 사용한다.

**명령어 실패시**

만약 위의 명령어가 권한 문제로 실패했다면 hadoop 계정과 hive 계정을 따로 사용하는 경우일 것이다.  
이때는 hive에서 사용하는 HDFS 디렉토리에 대해 hive에 사용권한을 아래와 같이 부여해야 한다.

*$ $HADOOP\_HOME/bin/hadoop fs -mkdir       /tmp*

*$ $HADOOP\_HOME/bin/hadoop fs -mkdir       /user/hive/warehouse*

*$ $HADOOP\_HOME/bin/hadoop fs -chmod a+w   /tmp*

*$ $HADOOP\_HOME/bin/hadoop fs -chmod a+w   /user/hive/warehouse*

#### 조회하기

TBD

#### 플럼과 연동하기

TBD

## 분석하기

TBD

Cascading??

awstats

주민번호 탐지

== HBase Quick Start

**HBase Quick Start**

원문: http://hbase.apache.org/book/quickstart.html

이 가이드 문서에서는 로컬 파일 시스템을 기반으로, HBase 인스턴스를 독립(standalone) 모드로 설정하는 방법을 설명한다. 또한 HBase 쉘을 사용해서 테이블을 생성하고, 행을 추가하는 방법을 설명한다. 그리고 HBase 인스턴스를  초기화하고, 중지시키는 방법도 설명한다. 아래에서 설명할 예제는 10분이면 충분히 따라할 수 있다(물론 HBase를 다운로드하는 시간은 뺴고 말이다).

**1.2.1 안정화된 가장 최신 버전의 릴리즈 다운로드하고, 압축 풀기**

[Apache Download Mirror](http://www.apache.org/dyn/closer.cgi/hbase/)s에서 다운로드할 수 있는 사이트 목록을 확인할 수 있다. 최상단에 있는, 권장하는 사이트를 클릭한다. 그러면 HBase 릴리즈를 다운로드할 수 있는 미러 사이트에 접속된다. stable 폴터를 클릭한 후, 확장자가 .tar.gz로 끝나는 파일을 로컬 파일 시스템으로 다운로드 한다. 여기에서는 hbase-0.95-SNAPSHOT.tar.gz를 다운로드했다.

다운로드한 파일을 압축을 풀고, tar를 해제한다. 그리고 나서, 압축이 풀린 디렉토리로 이동한다.

$ tar xfz hbase-0.95-SNAPSHOT.tar.gz  
$ cd hbase-0.95-SNAPSHOT

이렇게 하면, HBase를 실행할 준비는 끝난다. 하지만 실행하기 전에 먼저 conf/hbase-site.xml 파일을 수정해야 한다. conf/hbase-site.xml을 열어서 HBase가 저장할 디렉토리명을 hbase.rootdir 속성에 설정한다.

<?xml version="1.0"?>  
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>  
<configuration>  
  <property>  
    <name>hbase.rootdir</name>  
    <value>file:///DIRECTORY/hbase</value>  
  </property>  
</configuration>

이 설정에서 DIRECTORY에 해당하는 부분을 HBase가 데이터를 저장할 디렉토리로 변경한다. hbase.rootdir는 디폴트로 /tmp/hbase-${user.name}로 설정된다. 따라서 디폴트 속성을 그대로 두면, 서버를 리부팅할 때 지금까지 사용한 데이터가 날아가게 된다(대부분의 운영체제는 재시작할 때 /tmp를 날린다).

**1.2.2 HBase 실행하기**

이제 HBase를 실행해 보자.

$ ./bin/start-hbase.sh

이 명령어를 실행하면, 마스터(Master)가 기동된다. 이때 로그는 logs/hbase-user-master-example.org.out에 기록된다.

이제 HBase 인스턴스가 독립 모드로 실행된 상태다. 이처럼 HBase가 독립모드로 실행되면, HBase는 단일 JVM 위에서 모든 데몬, 즉 HBase 데몬과 ZooKeeper 데몬을 실행한다. HBase 로그는 하위 디렉토리인 log/에서 찾을 수 있다. HBase를 실행하는데 문제가 있다면, 로그를 찾아서 확인해 볼 수 있다.

***자바가 설치되어 있는가?***

앞에서 설명한 내용은 모두 여러분의 머신에 Oracle 버전의 자바가 설치되어 있고, 경로에 등록되어 있다고 전제한다. 자바가 제대로 설치되어 있는지, 그리고 경로가 제대로 설정되었는지 확인하려면 java 명령어를 실행해보라. 그러면 자바 프로그램에서 받는 파라미터에 대한 정보를 볼 수 있다(HBase는 6 버전의 자바를 필요로 한다). 만약 정보가 나오지 않는다면, HBase는 제대로 실행되지 않게 된다. 그러면 먼저 자바를 설치한 후, conf/hbase-env.sh 파일을 열어서 JAVA\_HOME 부분의 주석을 제거한다. 그리고 자박가 설치된 디렉토리로 적절하게 설정한다. 그런 후에 앞에서 설명한 부분을 다시 시도해 보라.

**1.2.3 쉘로 작업하기**

실행중인 HBase에는 쉘을 통해 접속할 수 있다.

$ **./bin/hbase shell**  
HBase Shell; enter 'help<RETURN>' for list of supported commands.  
Type "exit<RETURN>" to leave the HBase Shell  
Version: 0.90.0, r1001068, Fri Sep 24 13:55:42 PDT 2010  
  
hbase(main):001:0>

help라고 입력한 뒤, 엔터 키를 누르면 쉘 명령어 목록과 옵션을 확인할 수 있다. 도움말 맨 마지막 문단을 보면, HBase에서 변수나 명령어 인자를 어떻게 입력해야 하는지 확인할 수 있다. 특히 테이블 이름, 행, 컬럼 등을 인용부호로 감싸고 있다는 점을 유심히 봐야 한다.

test라는 이름으로 테이블을 생성해보자. 컬럼 패밀리는 cf라는 이름을 주고, 하나의 컬럼 패밀리만을 만들자. 테이블의 내요을 조회한 후, 몇 가지 값을 입력하면 테이블이 제대로 생성되었는지 확인해 볼 수 있다.

hbase(main):003:0>**create 'test', 'cf'**  
0 row(s) in 1.2200 seconds  
hbase(main):003:0>**list 'test'**  
..  
1 row(s) in 0.0550 seconds  
hbase(main):004:0> **put 'test', 'row1', 'cf:a', 'value1'**0 row(s) in 0.0560 seconds  
hbase(main):005:0> **put 'test', 'row2', 'cf:b', 'value2'**  
0 row(s) in 0.0370 seconds  
hbase(main):006:0> **put 'test', 'row3', 'cf:c', 'value3'**0 row(s) in 0.0450 seconds

이 예제에서 값 3개를 차례대로 추가했다. 처음에는 row1의 컬럼 cf:a에, 값으로는 value1을 입력했다. HBase에서 컬럼은 컬럼 패밀리를 가리키는 접두사(이 예제에서는 cf)와, 콜론, 그리고  컬럼 이름에 대한 접미사(이 경우에는 a)로 구성된다.

이제 데이터가 제대로 입력되었는지 확인해 보자.

아래와 같이, 테이블에 대해 scan 명령어를 실행하자.

hbase(main):007:0> **scan 'test'**  
ROW        COLUMN+CELL  
row1       column=cf:a, timestamp=1288380727188, value=value1  
row2       column=cf:b, timestamp=1288380738440, value=value2  
row3       column=cf:c, timestamp=1288380747365, value=value3  
3 row(s) in 0.0590 seconds

아래와 같이 한 행을 조회해보자.

hbase(main):008:0> **get 'test', 'row1'**  
COLUMN      CELL  
cf:a        timestamp=1288380727188, value=value1  
1 row(s) in 0.0400 seconds

이제 테이블을 비활성화한 후, 테이블을 삭제하자. 이렇게 하면 앞에서 한 작업들이 모두 초기화된다.

hbase(main):012:0> **disable 'test'**  
0 row(s) in 1.0930 seconds  
hbase(main):013:0> **drop 'test'**  
0 row(s) in 0.0770 seconds

exit 명령어를 입력해서, 쉘을 빠져 나온다.

hbase(main):014:0> exit

**12.4 HBase 중지하기**

stop 스크립트를 실행해서 HBase 인스턴스를 중지할 수 있다.

$ ./bin/stop-hbase.sh  
stopping hbase...............

**12.4 다음에 익혀야 할 내용들**

지금까지 HBase를 독립 모드로 설정하는 부분에 대해 설명했다. 이처럼 HBase를 독립모드로 실행하면, 테스트하거나 연습삼아 하는 경우에만 효과적이다. 이제 "2장. 설정하기"로 넘어가서 다양한 HBase 실행 모드에 대해 상세히 살펴보자. 2장에서는 HBase를 분산 모드로 설정할 때 필요한 요구사항과, 중요한 설정 정보들에 대해 설명한다.

## 관리하기

지금까지 몇 개의 맵리듀스 프로그램을 개발했다. 이러한 프로그램은 crontab을 이용하여 관리하수도 있겠지만, 이보다는 잡 관리에 특화된 프레임워크를 사용…

oozie

## 검색하기

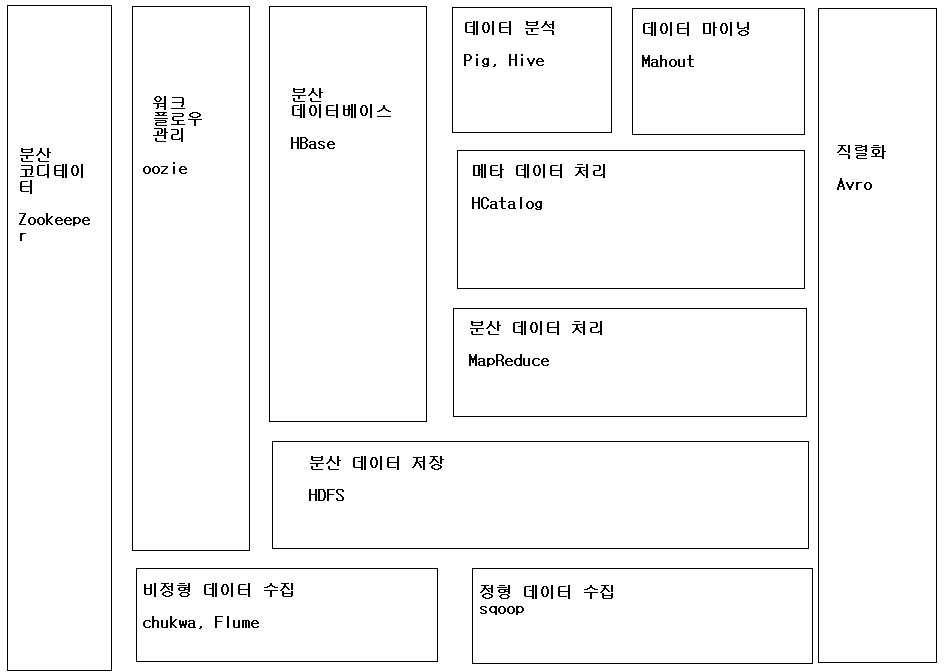
맵리듀스 프레임워크는 기본적으로 배치 방식

수분에서 수십분까지 걸림. 데이터 량에 의존적

수초에서 수십초 내에 간단한 처리를 위해서는 검색엔진을 도입

elasticsearch

# 2부 이해하기



## 플럼(Flume)

플럼은 분산 환경에서 대량의 로그 데이터를 수집하여 모은 후 전송하기 위한 솔루션이다. 플럼은 스트리밍 데이터 플로우(streaming data flow)를 기반으로 하며, 아키텍처가 단순하고 유연하다. 또한 간단하며 확장가능한 데이터 모델을 제공하여, 실시간 분석 애플리케이션 쉽게 개발할 수 있다.

플럼은 클라우데라(Cloudera)에서 0.9 버전까지 개발하였으며, 2012년에 플럼 NG(Next Generation)라는 코드명으로 아파치 정식 프로젝트로 등록되었다. 클라우데라에서 개발한 기존 버전은 OG(Original Generation) 버전이라고 부르며, 두 버전간의 차이점은 부록에서 자세히 설명한다. 이 책에서는 NG 버전이 아닌 OG 버전을 기준으로 설명한다. 하지만 기본 개념은 비슷하므로 OG에서 설명하는 내용은 NG를 사용할 떄도 도움이 되리라고 본다.

플럼은 단일 노드에 장애가 발생하더라도 운영할 수 있도록 설계되었다. 뿐만 아니라 로그 유실을 방지하기 위한 신뢰 수준을 상황에 맞게 설정할 수 있다. 또한 장애 발생시 다양한 복구 메커니즘을 제공한다.

특히 클러스터로 구성된 다양한 장치로부터 로그 파일들을 수집한 후, 하둡 분산 파일 시스템과 같은 중앙 저장소에 저장하는 로깅 시스템을 구축해야 할 때 가장 효과적인 솔루션이다.

### 개요

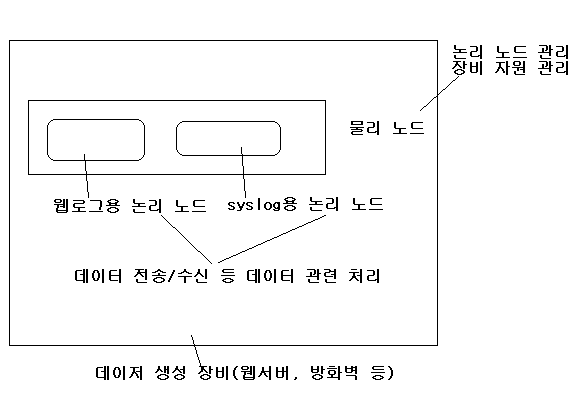
#### 물리 노드와 논리 노드

플럼을 설치한 후 기동하면 단일 JVM 인스턴스에서 단일 자바 프로세스로 실행된다. 이러한 플럼 자바 프로세스를 물리 노드(physical node)라고 부른다. 쯕 플럼을 설치한 장비는 반드시 하나의 물리 노드를 가지게 된다. 물리 노드는 데이터를 수집하는 역할을 직접 수행하지는 않는다.

이처럼 데이터를 수집한느 역할은 논리 노드(logical node)에서 담당한다. 하나의 물리 노드에서는 다수의 논리 노드를 실행할 수 있다. 예를 들어 한 장비에서 웹로그 뿐만 아니라 syslog까지 수집해야 하는 경우라면, 웹로그 파일을 전송하는 논리 노드와 syslog를 전송하는 논리 노드, 즉 총 2개의 논리 노드를 실행하여야 한다.

물리 노드는 논리 노드를 관리하는 컨테이너 역할을 하며, 장비의 자원을 각 논리 노드에 할당하는 일을 중재한다.

<그림. 물리 노드와 논리 노드와 관계>



#### 아키텍처

플럼의 아키텍처는 단순하며, 튼튼하고, 유연하다.  플럼 아키텍처는 스트림 중심의 데이터 플로우를 기반으로 한다. 데이터 플로우(data flow)란 하나의 데이터 스트림이 생성지에서 목표지로 전달되어 처리되는 방식이다. 각 데이터 플로우는 이벤트를 전송하고 수집하는 일련의 논리 노드(logical node)들로 구성된다. 즉 논리 노드를 서로 순차적으로 연결하여 데이터 플로우를 구성할 수 있다.

데이터 플로우를 구성하기 위하 논리 노드를 서로 연결하는 일은 모두 플럼 마스터에서 관리한다. 플럼 마스터는 플럼이 설치된 물리 노드와 논리 노드에 대한 정보를 저장하고 관리한다. 플럼을 설치한 후 물리 노드가 기동된 경우, 데이터를 수집하기 위한 설정은 플럼 마스터에서 수행한다. 이렇게 설정한 설정 정보 역시 플럼 마스터를 통해 물리 노드에 배포할 수 있으며, 설정 정보를 수신한 물리 노드는 설정 정보에 맞게 논리 노드를 기동하게 된다.

논리 노드는 차례대로 플럼 마스터와 주기적으로 hearbeat을 통신하여, 모니터링과 관련된 정보를 서로 공유한다.

아래 그림은 로그 데이터를 수집하기 위해 플럼을 사용하는 전형적인 레이어 구성이다.

|  |
| --- |
| http://www.gurubee.net/download/attachments/26739848/flume_architecture.png |

그림에서 보는 바와 같이 각 논리 노드는 에이저트 티어, 컬렉터 티어, 스토리지 티어, 총 3개의 티어로 구성된다.

데이터가 생성되는 장비에 설치하여 데이터를 전송하는 역할을 맡는 논리 노드를 에이전트 티어라고 부른다. 에이전트 노드는 생성된 데이터를 다음 티어인 컬렉터 노드로 전송하는 역할을 맡는다. 다시 말해 컬렉터 노드는 특정 포트로 데이터가 수신되기를 기다리고 있다. 그리고 데이터가 수신되면, 해당 데이터를 스토리지 티어로 다시 전송하게 된다.

예를 들면 에이전트는 syslog 데이터를 리스닝하거나 웹서버나 하둡 JobTracker와 같은 서비스에서 생성되는 로그를 모니터링하고 있는 머신일 수 있다. 에이전트는 데이터 스트림을 생성하여 컬렉터로 전송한다. 그러면 컬렉터는 데이터 스트림을 합친 후, HDFS와 같은 스토리지 티어에 효과적으로 저장할 수 있게 된다.

헷갈릴 수 있는 부분은 여기서 말한 노드는 모두 물리 노드가 아닌 논리 노드에 해당한다. 플럼은 유연하게 설계되었는데, 플럼을 설치한 후 물리 노드를 기동하는 방식은 에이전트 노드나 컬렉터 노드에 관계가 없다. 물리 노드에 어떠한 역할을 가진 논리 노드를 기동하느냐가 중요하다. 에이전트 성격의 논리 노드를 기동할 수도 있고, 컬렉터 성격의 논리 노드를 기동할 수도 있다.

에이전트 노드나 컬렉터 노드도 크게 차이나는 부분은 없다. 에이전트 노드는 데이터가 생성되는 출발지에 위치하여 커렉터 노드로 데이터를 전송하며, 컬렉터 노드는 에이전트가 전송하는 데이터를 다시 특정 티어로 전송한다는 점에서는 동일하다.

논리 노드는 매우 유연하게 추상화되었는데, 특히 각 논리 노드는 모두 source와 sink, 두개의 요소로 구성된다. source는 논리 노드가 수집할 데이터에 대한 설정이며, sink는 수집한 데이터를 전송할 대상에 대한 설정이다. 따라서 각 논리 노드는 source와 sink가 어떻게 설정되느냐에 따라 달라진다. 뿐만 아니라 각 논리노드를 서로 연결할 수도 있다. 수신한 데이터를 단순히 전송하는 대신 전처리 과정과 같은 중간 처리 과정이 필요하다면 데코레이터(decorator)를 사용할 수 있다.

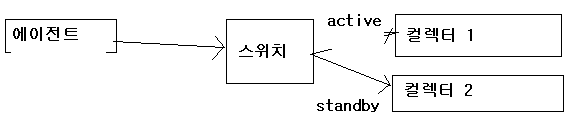
논리 노드가 어떤 데이터를 수신하여(source가 담당), 어디로 전송 또는 저장할지(sink가 담당)에 대한 설정은 마스터 노드에서 작업할 수 있다. 플럼을 설치하여 물리 노드를 기동한 상황이라면, 논리 노드를 설정하는 일은 런타임에 할 수 있다. 따라서 해당 장비에 로그인하지 않고도 마스터 노드를 통해 언제라도 설정 정보를 갱신할 수 있다.

뿐만 아니라 하나의 논리 노드는 여러 개의 sink를 가질 수 있는 fan-out 기능을 제공한다. fan-out을 이용하면 하나의 논리노드에서 다수의 데이터 흐름을 만들 수 있으며, 각 하위 흐름은 서로 다르게 설정할 수도 있다. 예를 들어 souce가 수신한 데이터를 HDFS에 저장하는 동시에, 스트리밍 형태로 실시간 분석을 해야 할 경우가 있다. 그러면 하둡 저장을 위한 sink와 실시간 분석을 위한 sink를 하나의 논리 노드에 순서대로 연결한다. 그러면 source가 수신한 데이터는 각 sink에 순서대로 전송되게 된다.

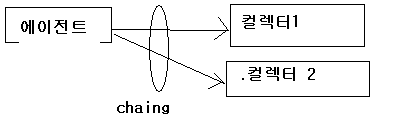
마지막으로 컬렉터 HA를 위한 체이닝(chaining) 기능을 제공한다. 다음 그림과 같이 하나의 에이전트에서 하나의 컬렉터로 데이터 흐름을 구성한다고 해보자.



만약 에이전트와 컬렉터 사이의 네트워크가 단절되거나 컬렉터가 다운된다면, 에이전트가 전송하는 로그가 유실될 가능성이 있다. 이와 같은 경우를 대비하기 위해 아래와 같이 스위치를 이용하여 HA를 구성하는 경우가 많다.



플럼은 스위치가 없이도 HA를 구성할 수 있는 체이닝 기능을 제공한다. 아래 그림과 같이 체이닝을 구성한 경우,



컬렉터 1에 로그를 전송할 수 없게 되면 자동으로 컬렉터 2로 로그를 전송하게 된다.

#### 특징

* **신뢰성(Reliability)**

플럼은 장애가 나더라도 로그를 유실없이 전송함을 보장한다. 대규모의 분산 시스템이라면 부분적인 장애가 나기 쉽상이다. 장비 자체가 물리적으로 고장날 수도 있고, 네트워크 대역이 막힐 수도 있으며, 메모리와 같은 컴퓨팅 자원이 부족해질 수 있다. 심지어는 소프트웨어가 느리게 실행되거나 심지어 다운될 수도 있다. 플럼은 fault-tolerance를 핵심 설계 원칙으로 내세운다. 즉 여러 구성요소가 다운되더라도 로그를 계속해서 수집할 수 있다. 다시 말해 에이저트 노드가 실행되는 상황이라면, 어떠한 경우에도 에이전트 노드가 수집한 로그를 결국에는 데이터 플로우의 종착역인 컬렉터 노드로 전송한다는 점을 보장한다.

하지만 이처럼 높은 신뢰 수준을 보장하려면 처리 속도가 높은 값비싼 장비를 사용해야 하거나, 또는 여러대의 장비를 구비해야 한다. 하지만 대개의 경우 실제 필요로 하는 신뢰 수준은 이보다 더 낮은 경우가 많다. 따라서 플럼은 데이터 플로우별로 신뢰 수준을 사용자가 직접 설정할 수 있는 기능을 제공한다. 플럼에서는 세가지 종류의 신뢰 수준은 다음과 같다.

* E2E(End-to-End)

에이전트가 기동중이라면 에이전트에서 수집한 이벤트를 마지막 목적지까지 반드시 전송한다. 이러한 신뢰 수준을 사용하는 경우, 에이전트는 먼저 이벤트를 디스크의 write-ahead log(WAL)에 기록한다. 따라서 에이전트가 다운되어 재시작되더라도, 어디까지 이벤트를 전송했는지 알 수 있다. 이벤트가 성공적으로 데이터 플로우의 끝지점까지 전송되면, 이벤트를 전송한 에이전트로 ack이 반송된다. 이를 통해 디스크에 기록된 이벤트가 더이상 전송할 필요가 없다는 사실을 판단한다. 이정도의 신뢰도를 설정하게 되면, 에이전트에 장애가 나는 대다수의 상황에도 끄떡없다.

단점은 초당 이벤트 처리 속도(EPS)가 떨어지게 된다. 따라서 신뢰 수준을 낮출 수 없는 상황에서 요구되는 EPS를 맞추기 위해서는 더 비싼 장비를 구비해야 한다.

* SoF(Store on Failure)

SoF 레벨에서는 한 홉(hop)까지만의 ack을 요구한다. 따라서 이벤트를 전송하는 노드가 장애를 인지하게 되면, 데이터를 로컬 디스크에 기록해두고, 다음 노드가 복구될 때까지 이벤트를 저장하거나 또 다른 노드로 전송하게 된다. 이 레벨에서는 복합적인 장애가 발생하게 되면 데이터가 유실될 수도 있다. SoF 레벨은 DFO(Disk Fail Over) 라고도 부른다.

* BE(Best Effort)

BE 레벨에서는 다음 홉에 데이터를 전송하며, ack을 받거나 재시도하는 처리를 전혀 하지 않는다. 따라서 노드가 다운되면, 전송중이거나 전송받는 중이었던 데이터는 유실될 수 있다. BE 레벨이 신뢰도가 가장 낮은 레벨이지만, 가장 가벼우며 EPS도 가장 높다.

* **확장성(Scalability)**

노드 추가 및 제거가 용이하다. 예를 들어 한 컬렉터 서버에서 1000EPS를 처리하며, 현재 3대의 컬렉터 서버를 운영중이라고 해보자. 로그를 수지해야할 대상이 증가하여, 추가적으로 700EPS를 처리해야 한다고 하자.

플럼은 선형 확징(scale out)이 가능하도록 설계되었다. 따라서 기존의 컬렉터 서버와 유사한 성능의 장비를 클러스터에 쉽게 추가할 수 있다.

* **운영가능성(Manageability)**

마스터 노드를 통해 클러스터에 분산된 여러 노드를 직접 로그인 하지 않고도 원격에서 설정 및 모니터링할 수 있다. NG와의 가장 큰 차이가 바로 이 부분이다.

* **모듈 확장성(Extensibility)**

플럼은 플러그인 형태의 데이터 모델을 지원하도록 만들어졌다. 따라서 필요로 하는 기능을 플럼에서 제공하지 않더라도, 데코레이터 등을 활용해서 새로운 기능을 쉽게 추가할 수 있다.

### 설치

하둡과 마찬가지로 플럼은 독립 모드, 의사 분산 모드, 완전 분산 모드로 설치할 수 있다.

### 의사 분산 모드로 설치하기

1부 적용하기에서 이미 의사 분산 모드로 설치하였다.

### 완전 분산 모드로 Flume 설치하기

[[ 작업중 ]]

### 고급 기능 사용하기

#### 체이닝

#### fan-out

#### 데코레이터 개발하기

데코레이터 예제코드

메타데이터 추가 데코레이터

**DataTypeTaggingDecorator.java**

public class DataTypeTaggingDecorator <S extends EventSink> extends EventSinkDecorator<S> {

static final Logger LOG = LoggerFactory.getLogger(DataTypeTaggingDecorator.class);

private String type;

private byte[] typeBytes;

private String cid;

private byte[] cidBytes;

private String ip;

private byte[] ipBytes;

private String domain;

private byte[] domainBytes;

public DataTypeTaggingDecorator(S s, String... argv) {

super(s);

this.type = argv[0];

this.typeBytes = type.getBytes();

this.cid = argv[1];

this.cidBytes = cid.getBytes();

this.ip = argv[2];

this.ipBytes = ip.getBytes();

if( argv.length > 3 ) {

domain = argv[3];

domainBytes = domain.getBytes();

}

}

@Override

public void append(Event e) throws IOException, InterruptedException {

e.set("type", typeBytes);

e.set("cid", cidBytes);

e.set("aip", ipBytes);

e.set("domain", domainBytes);

// e.set("seq", (ip+"\_"+(seqNum++)).getBytes());

super.append(e);

}

public static SinkDecoBuilder builder() {

return new SinkDecoBuilder() {

@Override

public EventSinkDecorator<EventSink> build(Context context, String... argv) {

if (argv.length < 3) {

throw new IllegalArgumentException("usage: dataTypeTaggingDeco(<type name>,<cid>,<aip>[,domain] )");

}

return new DataTypeTaggingDecorator<EventSink>(null, argv);

}

};

}

public static List<Pair<String, SinkFactory.SinkDecoBuilder>> getDecoratorBuilders() {

return Arrays.asList(new Pair<String, SinkFactory.SinkDecoBuilder>("dataTypeTaggingDeco", builder()));

}

}

아래는 위의 데코레이터를 사용해서 설정하는 명령어 샘플이다.

|  |
| --- |
| http://www.gurubee.net/download/attachments/26739848/flume_config_decorate_sample.png |

인코딩 변환 데코레이터

**EncodingConvertDecorator.java**

package com.samsung.sds.beaver.flumeplugin.decorator;

import java.io.IOException;

import java.util.Arrays;

import java.util.List;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.cloudera.flume.conf.Context;

import com.cloudera.flume.conf.SinkFactory;

import com.cloudera.flume.conf.SinkFactory.SinkDecoBuilder;

import com.cloudera.flume.core.Event;

import com.cloudera.flume.core.EventImpl;

import com.cloudera.flume.core.EventSink;

import com.cloudera.flume.core.EventSinkDecorator;

import com.cloudera.util.Pair;

public class EncodingConvertDecorator <S extends EventSink> extends EventSinkDecorator<S> {

static final Logger LOG = LoggerFactory.getLogger(EncodingConvertDecorator.class);

private String sourceEnc;

private byte[] sourceEncBytes;

private String targetEnc;

private byte[] targetEncBytes;

public EncodingConvertDecorator(S s, String... argv) {

super(s);

this.sourceEnc = argv[0];

this.sourceEncBytes = sourceEnc.getBytes();

this.targetEnc = "UTF-8";

if( argv.length > 1 ) {

this.targetEnc = argv[1];

}

this.targetEncBytes = targetEnc.getBytes();

}

@Override

public void append(Event e) throws IOException, InterruptedException {

e.set("sourceEnc", sourceEncBytes);

e.set("targetEnc", targetEncBytes);

e = new EventImpl((new String(e.getBody(),sourceEnc)).getBytes(targetEnc), e.getTimestamp()

, e.getPriority(), e.getNanos(), e.getHost(), e.getAttrs());

super.append(e);

}

public static SinkDecoBuilder builder() {

return new SinkDecoBuilder() {

@Override

public EventSinkDecorator<EventSink> build(Context context, String... argv) {

if (argv.length < 1) {

throw new IllegalArgumentException("usage: encCvtDeco(sourceEncoding[, targetEncoding=\"UTF-8\"]))");

}

return new EncodingConvertDecorator<EventSink>(null, argv);

}

};

}

public static List<Pair<String, SinkFactory.SinkDecoBuilder>> getDecoratorBuilders() {

return Arrays.asList(new Pair<String, SinkFactory.SinkDecoBuilder>("encCvtDeco", builder()));

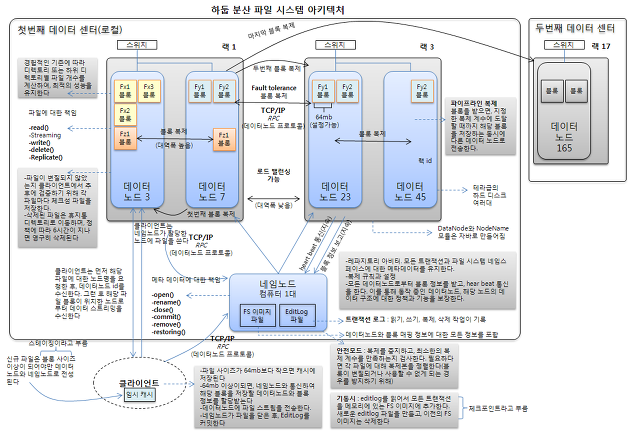
}

}

## 하둡 분산 파일 시스템(HDFS)

### 하둡 파일 시스템 구성

하둡은 분산 파일 시스템(HDFS)과 분산 컴퓨팅을 위한 맵리듀스(MapReduce)로 구성된다. 하둡 파일 시스템은 구글 파일 시스템의 설계와 아이디어를 이용하여 개발되었으며, 구글 파일 시스템과 많이 유사하다.



하둡 파일 시스템은 하나의 네임 노드 서버, 세컨데리 네임 노드 서버, 그리고 다수의 데이터 노드 서버로 구성된다.

#### 네임 노드 (Name Node)

하둡 파일 시스템은 파일을 분산 저장하고, 프로그램을 분산/병렬 처리하기 위해 마스터/슬레이브 구조를 가진다. 이때 네임 노드가 마스터 역할을 하며, 데이터 노드가 슬레이브 역할을 맡는다.

네임 노드는 파일 시스템의 네임 스페이스(디렉토리, 파일명, 파일 블록 등)을 관리한다. 클라이언트는 파일 시스템에 접근하기 위해 네임 노드에 요청을 하며, 네임 노드는 클라이언트에게 파일이 위치한 데이터 노드 목록을 반환한다. 그러면 클라이언트는 해당 데이터 노드에 직접 접속하여 파일에 접근한다.

주의할 점은 네임 노드는 나머지 노드와는 달리 이중화 구성이 되지 않아 단일 장애 지점이 된다는 사실이다.

#### 데이터 노드(Data Node)

데이터 노드는 파일에 대한 실질적인 I/O 처리를 담당한다. 예를 들어 파일을 하둡 파일 시스템에 추가한 경우, 해당 파일은 먼저 블록 단위로 나뉜 후, 여러 데이터 노드에 분산되어 저장된다. 또한 특정 노드에 장애가 발생하더라도 데이터 유실을 방지하기 위해, 각 블록은 복제되어 다른 데이터 노드에 저장된다.

예를 들어 4개의 데이터 노드를 가진 하둡 클러스트터에서, /user/socurites/ 디렉토리에 129MB인 foo.data 파일을 추가한다고 해보자. 그러면 foo.data 파일은 우선 블록 단위(기본적으로 64MB)로 분할되어 저장된다.

[[ 설명하기 어려운데.. 그림 필요 ]]

각 데이터 노드는 자신의 파일 블록 정볼르 네임 노드에게 지속적으로 전송한다.

#### 세컨데리 네임 노드(Secondary Name Node)

세컨데리 네임 노드는 네임 노드에서 관리하는 네임 스페이스의 정보를 백업받아 그 당시의 스냅샷을 생성하는 기능을 수행한다. 자세한 내용은 다음 장에서 설명한다.

#### 파일 생성 동작 방식

하둡 파일 시스템에 파일을 저장할 때, 각 구성 요소가 동작하는 방식은 다음과 같다.

[그림. 파일 생성시 하둡 파일 시스템의 동작 방식]

1. 클라이언트는 네임 노드에게 파일 생성을 요청한다.
2. 파일 생성 요청을 받은 네임 노드는 메모리 상에 해당 파일의 경로 정보를 생성한다. 그리고 다른 클라이언트에서 해당 경로로는 파일을 생성하지 못하도록 락을 건다.
3. 네임 노드는 파일을 실제로 저장할 데이터 노드 목록을 클라이언트로 반환한다.
4. 데이터 노드 목록을 받은 클라이언트는 첫 번째 데이터 노드로 파일을 전송한다. 이때 데이터 노드의 목록을 데이터의 첫부분에 함께 전송한다.
5. 첫 번째 데이터 노드는 자신의 로컬 디스크에 파일을 저장한 후, 목록의 다음에 있는 데이터 노드에 파일을 전송한다. 두 번째 데이터 노드 또한 자신의 로컬 디스크에 파일을 저장한 후, 세 번째 데이터 노드로 파일을 전송한다. 이러한 방식으로 복제 계수만큼 데이터 노드로 전송되게 된다. 일반적으로 복제 계수로 3을 사용하므로, 대개의 경우 세 번쨰 데이터 노드까지 파일이 전송된다.
6. 만약 파일이 블록 크기(일반적으로 64MB)보다 클 경우, 클라이언트는 새로운 데이터 노드를 네임 노드에게 요청한다.
7. 저장이 완료되면 네임 노드는 메모리에 임시로 저장되어 있던 파일의 경로 정보를 메모리상에 관리하는 네임 스페이스로 이동시킨다. 그리고 네임 노드를 재시작한 후에도 경로 정보가 영구적으로 존재하도록, 네임 노드의 로컬 디스크에 파일 생성과 관련된 로그를 edits 파일에 기록한다.
8. 6번에서 클라이언트가 요청한 새로운 데이터 노드 목록을 전달한다.
9. 클라이언트는 4~6번 과정을 반복한다.
10. 파일 전송이 완료되면, 클라이언트는 close() 명령을 네임 노드로 전달한다.

[참고]

일반적인 파일 시스템은 쓰기 연산을 하면, 물리적인 디스크에 바로 반영이 된다. 따라서 클라이언트에 장애가 발생하더라도, 장애가 발생하기 직전까지의 데이터는 파일 시스템에 저장되어 있다. 반면 하둡 파일 시스템에서는 파일이 일정 크기(블록 사이즈)를 넘어서거나, sync() 명령을 호출한 경우에만 파일이 물리적으로 저장된다.

#### 세컨데리 네임 노드 동작 방식

세컨데리 네임 노드는 네임 노드로부터 주기적으로 edits 파일을 다운로드 한후, 이전의 파일 네임스페이스에 대한 스냅샷인 fsimage 파일과 병합하여, 새로운 스냅샷인 fsimage 파일을 생성한다.

새롭게 만들어진 fsimage 파일을 네임 노드로 전송한다.

네임 노드는 기존의 fsimage 파일을 새로운 파일로 대체한 후, 비어 있는 edits 파일을 새로 생성한다.

네임 노드를 재시작할 경우, 최신의 fsimage 파일을 읽어 메모리로 로드한 후, edits 파일에 기록된 내역을 하나씩 수행한다.

이러한 방식을 통해 네임 노드 재기동 시간을 줄일 수 있다. 만약 fsimage 파일을 주기적으로 생성하지 않았다면, edits 파일의 사이즈가 엄청나게 클 것이며, 결국 변경사항을 반영하느라 많은 시간이 소모 될 것이다.

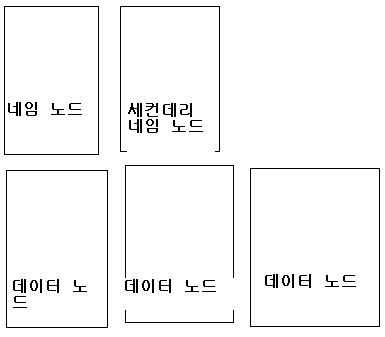
이러한 세컨데리 네임 노드의 동작 방식으로 인해, 세컨데리 네임 노드는 백업 노드로도 활용할 수 있다. 여기에 대해서는 네임 노드 백업 및 HA 구성을 살펴보라.

### 설치

#### 설치 요구 사항

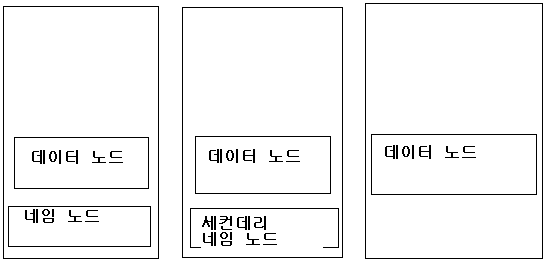
하둡 파일 시스템 설치를 위해서는 아래와 같이 기본적으로 5대의 리눅스 장비가 필요하다.

* 네임 노드
* 세컨데리 네임 노드
* 데이터 노드 3대



하지만 이 책을 읽는 시점에 운영할 수 있는 장비가 모자를 것이고, 나 또한 마찬가지다. 최소한의 머신을 구비하여 하둡 파일 시스템을 운영하려면 의사 분산 모드로 설치해볼 수 있다. 하지만 분산 환경의 장점을 완전히 누리자면 최소한 5대의 장빈느 필수적이다.

그렇긴해도 머신을 구비하기 어려운 독자를 위해 이 책에서는 아래와 같은 구성으로 총 3대의 머신만을 사용하고자 한다. 이렇게 하더라도 기본적인 사용법은 동일하다.



하둡 파일 시스템은 자바로 개발되었으며, 리눅스 환경에서만 기동된다. 또한 하나의 노드에서 하둡 클러스터에 있는 모든 노드를 관리(실행/중지) 할 수 있는 쉘 스크립트 명령어를 제공한다. 이러한 명령어는 모두 ssh를 기반으로 하므로, 모든 노드엔느 sshd가 설치하여 실행해야 한다.

따라서 각 머신은 아래와 같은 환경으로 구성되어 있어야 한다.

* 운영환경 : 리눅스
* JDK 1.6 이상
* sshd

#### ssh 환경 설정

[[ 1부 적용하기에서 설명했는데, 좀 더 자세히 설명할 필요가 있을까? ]]

#### 다운로드

하둡을 다운로드한다. 하둡은 아파치에서 제공하는 기본 배포판 이외에도 여러 배포판이 존재한다. 자세한 내용은 부록을 참고하라.

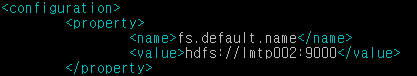
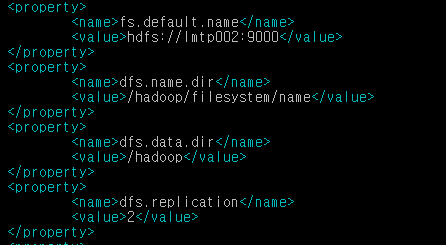
여기에서는 아피치에서 제공하닌 기본 배포판을 다운로드한다.

#### 환경 설정

하둡 클러스터는 수십 또는 수백대의 장비로 구성되기 마련이다. 따라서 각 장비별로 환경 설정은 하기란 쉬운 일이 아니다. 하둡은 네임 노드에서 모든 환경 설정을 한 후, 나머지 장비에 설정 파일을 배포하는 방식으로 사용된다. 즉 클러스터에 있는 모든 노드에서 동일한 환경 설정 파일을 공유하는 방식이다. 그러면 네임 노드에서는 네임 노드와 관련된 환경 설정 정보를 사용하여 기동되고, 데이터 노드에서는 데이터 노드와 관련된 환경 설정 정보를 사용하여 기동된다.

대개의 경우 유사한 머신으로 하둡 클러스터를 구성하는 경우가 많기 때문에 동일한 설정 정보를사용하더라도 크게 문제가 없다. 만약 장비의 스펙이 크게 차이가 나는 경우라면, 각 노드마다 환경 설정을 별도로 해야만 한다.

하둡의 환경 설정 파일은 하둡이 설치된 디렉토리의 conf/ 디렉토리에 있다. 하둡과 관련된 설정 파일은 다음과 같다.

* hadoop-env.sh  
  - 하둡이 실행하는 모든 프로세스에 적용되는 시스템 환경 값에 대한 스크립트  
  - 하둡에서 제공하는 모든 스크립트는 모두 이 스크립트를 먼저 실행한다.  
  - 하둡은 환경 변수를 시스템 변수로 등록하는 방법 대신, 이 파일에 환경 변수를 등록하는 방식이다.  
  - 필수적으로 JAVA\_HOME을 설정해야 한다.  
    
  
* core-site.xml  
  - 하둡 파일 시스템과 맵리듀스 프레임워크에서 공통적으로 사용할 환경 변수를 설정한다.  
  - 기본값은 src/core/core-default.xml에서 확인할 수 있다.  
  - 필수적으로 네임 노드에 대한 URI(fs.default.name)을 설정해야 한다.  
  
* hdfs-site.xml  
  - 하둡 파일 시스템에 대한 환경 변수를 설정한다.  
  - 기본 값은 src/hdfs/hdfs-default.xml에서 확인할 수 있다.  
    
    
  - dfs.name.dir  
  네임 노드에서 파일의 디렉토리 정보와 파일 정보 등을 저장하는 디렉토리  
  - dfs.data.dir  
  데이터 노드에서 데이터 파일이 저장되는 위치. 디스크가 여러 개인 경우 콤마(,)로 구분하여 기술  
  - dfs.replication  
  복제 계수. 기본 값은 3.
* mapred-site.xml  
  - 하둡 맵리듀스에 대한 환경 변수를 설정. 다음 장에서 자세히 설명한다.
* masters  
  세컨데리 네임 노드가 실행될 서버.
* slaves  
  데이터 노드가 실행될 서버  
  masters 파일과 slaves 파일에 등록된 노드는 하둡 파일 시스템 시작시 사용되며, 실행 중에는 사용되지 않는다.
* hadoop-metrics.properties  
  글쎄.. 설명해야 하나??

#### 배포

[[bin/slaves.sh 사용 방법 설명]]

### 실행

하둡 파일 시스템은 네임 노드 서버에서 실행한다. 최초 실행시에는 파일 시스템을 포맷해야 한다. 하둡 포맷 명령은 서버의 파일 시스템을 포맷하는 것이 아니라, 하둡 파일 시스템의 네임 스페이스 이미지 정보를 초기화하는 작업이다. 즉 dfs.name.dir에 설정한 디렉토리의 파일을 초기화한다.

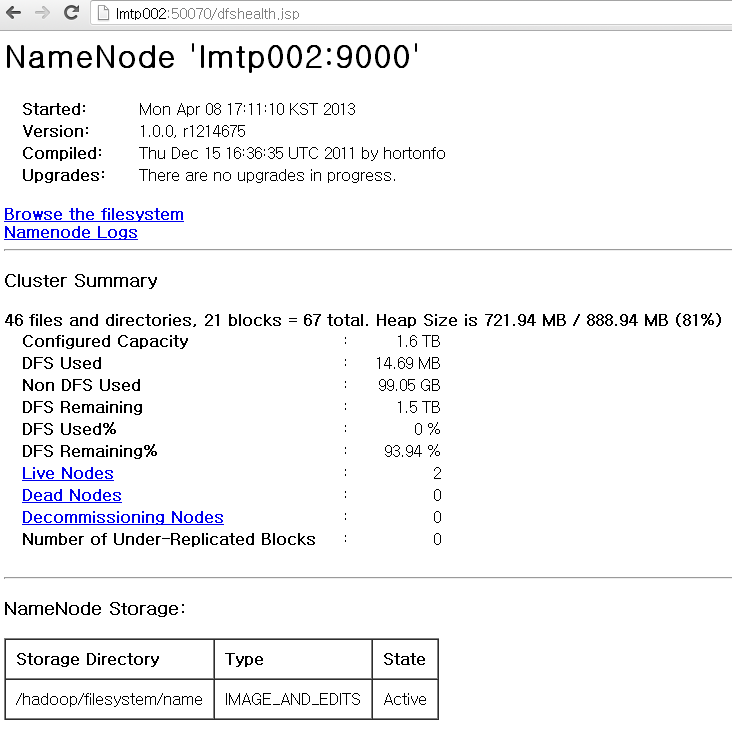
bin/hadoop namenode –format

하둡 파일 시스템을 실행한다.

bin/start-dfs.sh

하둡은 파일 시스템을 모니터링할 수 있는 웹 화면을 제공한다. 웹 화면은 <http://namenode_hostnam:50070> 으로 접속한다.

[그림. 하둡 파일 시스템 웹 모니터링 화면]



쉘에서는 bin/hadoop 명령어로 하둡 파일 시스템 관련 명령어를 수행할 수 있다. 아래는 하둡 파일 시스템 관리와 관련된 명령어다.

* namenode –frmat
* secondarynamenode
* namenode
* datanode
* dfsadmin
* fsck
* fs
* balancer
* jar

#### 안전 모드(safe mode)

하둡 파일 시스템을 기동하면 네임 노드는 fsimage 파일과 edits 로그 파일을 읽어 메모리 상에 파일 네임스페이스를 구성한다. 그리고 각 데이터 노드로부터 블록 정보를 수신한다. 네임 노드는 데이터 노드로부터 받은 블록 정보와 메모리의 파일 네임 스페이스 정볼르 비교해서 검증 작업을 수행한다. 이처럼 네임 노드가 기동되어 검증작업을 수행하는 상태를 안전 모드(safe mode)라고 부른다. 안전 모드 상태에서 하둡 파일 시스템은 읽기 모드로 동작한다.

### 하둡 파일 시스템 탐색

쉘에서 하둡 파일 시스템을 탐색할 때 쓰는 명령어 bin/ 디렉토리에 위치하며, 아래와 같은 형식을 가진다.

bin/hadoop fs –명령어 <인자>

하둡 파일 시스템 명령어는 유닉스 명령어와 비슷하지만 완전히 같지는 않다. 예를 들어 파일 목록을 보기 위해서는 ls 명령어를 사용한다.

bin/hadoop fs –ls /

먼저 하둡 파일 시스템에 새로운 디렉토리를 생성해 보자. 디렉토리를 생성하는 명령어는 mkdir이다.

bin/hadoop fs –mkdir /tmp/foo

디렉토리가 정상적으로 생성되었는지 확인해 보자.

bin/hadoop fs –ls /tmp

Found 1 items

drwxr-xr-x - hadoop supergroup 0 2013-05-03 14:10 /tmp//foo

이제 로컬 파일 시스템에 있는 test.txt 파일을 /tmp/foo 디렉토리에 추가해 보자.

bin/hadoop fs –put ./test.txt /tmp/foo

bin/hadoop fs –ls./test.txt /tmp/foo

Found 1 items

-rw-r--r-- 3 hadoop supergroup 0 2013-05-03 14:15 /tmp/foo/test.txt

유닉스 파일 시스템과 마찬가지로 디릭터리/파일 여부, 접근권한 및 소유자, 소유 그룹 등의 정보까지 확인할 수 있다. 하둡 분산 파일 시스템은 추가적으로 파일의 복제수(이 경우에는 “3”)까지 함께 출력한다. 앞서 본 디렉토리의 경우 복제라는 개념이 없으므로 “-“로 출력된다.

이제 반대로 하둡 파일 시스템에 있는 파일을 로컬 파일 시스템으로 가져와 보자.

bin/hadoop fs –get /tmp/foo/test.txt ./test\_copy.txt

또한 하둡 파일 시스템의 파일을 로컬로 가져 오지 않더라도 콘솔로 출력해서 볼 수도 있다.

bin/hadoop fs -cat /tmp/foo/test.txt

마지막으로 파일을 삭제해 보자.

bin/hadoop fs –rm /tmp/foo/test.txt

하둡 파일 시스템과 관련한 자세한 목록은 도움말을 통해 확인할 수 있다.

bin/hadoop fs –help

특정 명령어에 대한 도움말을 보고 싶다면 다음과 같이 확인할 수 있다.

bin/hadoop fs –help cat

### 파일 시스템 API 활용

하둡 파일 시스템은 콘솔에서 탐색할 수 있을 뿐만 아니라, 자바 프로그램에서 하둡 파일 시스템에 접근할 수 있는 API도 제공한다.

[[ 하둡 파일 시스템 API 설명 및 관련된 샘플 프로그램 설명하기 – 쉘의 명령어와 동일한 역할을 하는 간단한 샘플 프로그램 개발해서 설명할 것]]

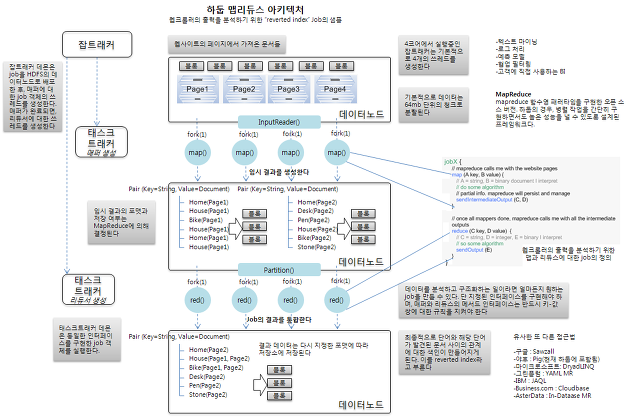
## 맵리듀스(MapReduce)

맵리듀스는 대용량의 데이터를 병렬로 처리하기 위한 소프트웨어인 동시에, 많은 저가의 장비로 페타바이트급의 데이터를 쉽게 처리하기 위한 일종의 프로그래밍 모델이기도 하다.

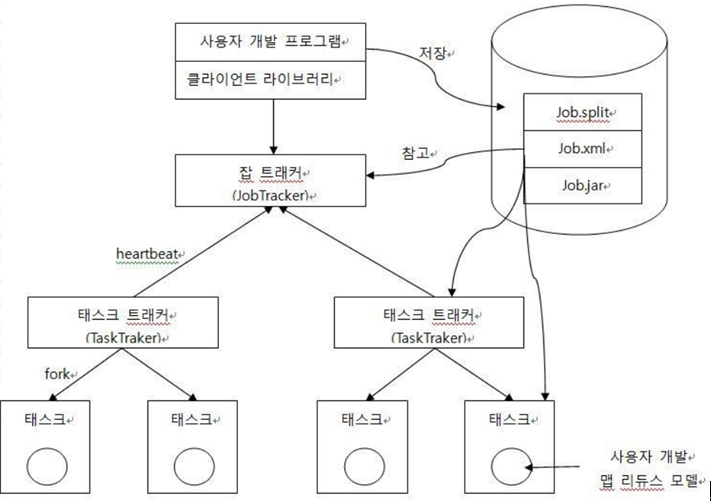
### 맵리듀스 시스템 구성

하둡 분산 파일 시스템과 맵리듀스 시스템을 합쳐서 하둡이라고 부른다. 하둡 맵리듀스 또한 분산환경으로 구성되며, 마스트/슬레이브 구조로 이루어져 있다. 잡 트래커가 마스트 역할을 하며, 태스크 트래커가 슬레이브 역할을 맡는다.

[그림. 하둡 맵리듀스 시스템 구성(상세)]



[그림. 하둡 맵리듀스 시스템 구성(간단)]



#### 잡 트래커(Job Tracker)

잡 트래커는 사용자가 개발한 맵 리듀스 잡(MapReduce Job) 프로그램을 관리한다. 예를 들어 사용자가 잡을 실행한 경우, 잡 트래커는 해당 잡을 태스크로 나누고, 어느 태스크 트래커에서 실행할 지 결정한다. 또한 실행 중인 태스크를 모니터링하며, 실패한 태스크가 있을 경우 자동으로 재실행 하기도 한다.

잡 트래커는 별도의 머신에서 독립적으로 실행할 수도 있지만, 일반적으로 네임 노드가 실행된 서버에서 함꼐 구동한다.

#### 태스크 트래커(Task Tracker)

잡 트래커가 할당한 태스크를 실제로 실행하는 데몬으로, 일반적으로 데이터 노드가 실행된 서버에서 함께 구동된다.

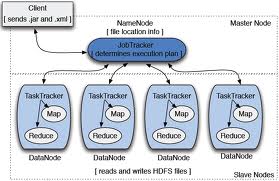
#### 잡 실행 과정

사용자가 맵 리듀스 잡을 실행하면, 잡 트래커는 아래와 같은 과정을 거쳐서 잡을 완료한다.

* 사용자는 잡 트래커에게 잡에 대한 실행 요청을 한다.
* 잡 트래커는 해당 잡에서 필요로 하는 입력 파일을 하둡 파일 시스템에서 읽은 후, 잡을 태스크 단위로 분리한다.
* 잡 트래커는 태스크 트래커별로 처리할 태스크 목록을 구성한다.
* 태스크 트래커는 주기적으로 heartbeat을 잡 트래커로 전송하며, 잡 트래커는 이 메시지의 반환 값에 처리할 태스크 아이디를 반환한다.
* 작업 아이디를 받은 태스크 트래커는 해당 태스크과 관련된 정보를 하둡 파일 시스템에서 로컬로 다운로드한다.
* 태스크 트래커는 fork 명령어를 수행해서 맵 또는 리듀스 태스크를 실행한다. 하나의 태스크 트래커는 동시에 여러 개의 맵이나 리듀스 태스크를 수행할 수 있다.

태스크 트래커는 잡 트래커와 주기적으로 통신하며, 만약 태스크 트래커로부터 설정된 시간 내에 통신이 되지 않는다면, 잡 트래커는 해당 태스크 트래커가 장애가 난 것으로 판단한다. 그리고 해당 태스크 해당 태스크 트래커에서 수행 중이던 작업을 다른 태스크 트래커에 할당해서 태스크를 재시작한다.

[그림. 잡 트래커와 태스크 트래거의 상호 작용]



### 맵리듀스 프로그램

#### 맵 리듀스 프로그램 모델

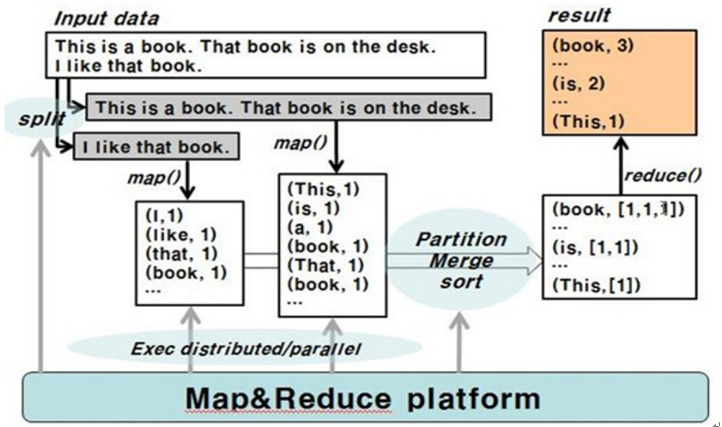
맵리듀스 프로그램 모델이란 LISP와 같은 하숨 기반 언어에서 이미 사용해오던 프로그램 모델이다. 간단히 도식화하면 다음과 같다.

[그림. 맵리듀스 프로그램 모델 개념도]



* 맵 함수의 입력으로 키(k1)와 값(v1)이 전달된다.
* 맵 함수는 출력으로, 새로운 키(k2)와 값(v2)의 목록을 생성한다.
* 입력에 대해 맵 함수가 반복적으로 수행되며, 그에 따라 여러 개의 출력 데이터가 생성된다.
* 맵 함수의 출력 데이터를 키(k2)로 정렬하면, 각 키에 대해 여러 개의 데이터가 존재하게 된다.
* 리듀스는 입력으로, 앞의 정렬된 키(k2)와 각 키에 대한 값 목록(list(v2))가 전달된다.
* 리듀스는 출력으로, 여러 개의 값(list(v3))를 생성한다.

아래는 문서에 존재하는 단어별 개수를 세는 맵리듀스 프로그램이다.



* 맵 함수는 입력으로 입력 파일의 라인 번호(키)와 해당 라인의 텍스트(값)을 전달받는다.
* 맵 함수는 공백으로 라인을 분리한 후, 어미와 조사 등을 제거해 단어를 생성한 후, 각 단어(키)와 단어의 출현 횟수인 1(값)을 출력으로 생성한다.
* 맵 함수의 출력 결과를 키(단어)를 기준으로 정렬한 후, 각 키 별로 단어의 출현 횟수를 목록으로 병합한다.
* 리듀스 함수는 단어(키)와 해당 단어의 출현 빈도 목록을 값으로 전달 받는다.
* 리듀스 함수는 출현 빈도 목록에 포함된 횟수에 대해 합계를 구한 후, 합계를 출력으로 생성한다.

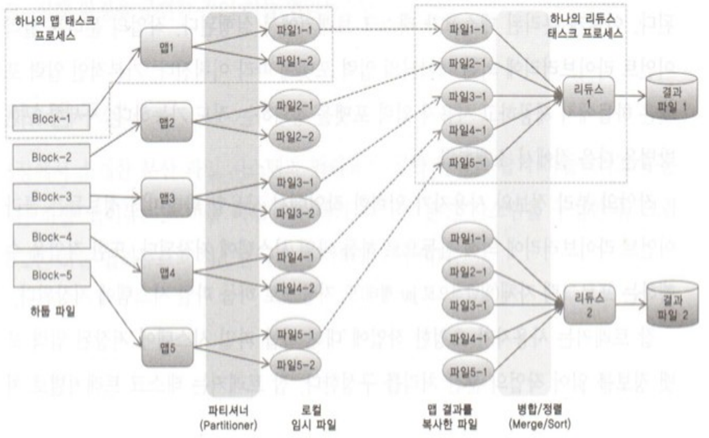
이 과정에서 입력 데이터에 대해 맵 함수는 병렬적으로 동시에 처리할 수 있다. 또한 리듀스도 각 단어에 대한 합계를 계산하는 일을 병렬적으로 수행할 수 있다. 이러한 프로그램 구조로 인해 멥리듀스 프로그램 모델은 분산/병렬 처리하기에 좋은 구조가 된다.

이러한 맵리듀스 프로그램을 만들 때, 하둡 맵리듀스 프레임워크는 아래와 같은 역할을 수행한다. 따라서 프로그래머는 단순히 맵함수와 리듀스 함수만을 구현하는 데만 집중할 수 있게 된다.

* 입력 파일을 라인 단위로 분할해서 맵 함수에 전달한다.
* 맵 함수를 분산된 서버에서 수행한다.
* 맵 함수의 출력 결과를 정렬/병합한다.
* 분산 처리된 맵 결과를 리듀스가 수행될 서버로 전송한다.
* 정렬/병합된 결과를 리듀스에 전달한다.

#### 맵리듀스 프로그램 처리 흐름

이제 맵리듀스 프로그램의 입출력 결과를 중심으로, 실제 데이터가 맵 태스크 또는 태스크 트래거 사이에 어떻게 이동하는지 좀더 자세히 살펴보자.



* 보통 하둡 파일 시스템에 저장된 데이터를 입력으로 받는다. 하둡 파일 시스템은 하나의 파일을 블록 단위(일반적으로 64MB)로 자동으로 분할한 후, 복젝 계수만큼 여러 노드에 분산되어 저장한다.
* 입력 파일의 블록 계수(복제본 제외)만큼 맵 태스크가 생성되며, 각 맵 태스크는 하나의 블록을 전달받는다.
* 맵리듀스 프레임워크가 입력 블록을 읽은 후, 레코드 단위(일반적으로 하나의 라인)로 맵 함수에 전달한다. 이처럼 입력을 읽어서 맵 함수에 전달하는 역할은 InputFormat 클래스가 담당한다.
* 각 레코드에 대해 맵 함수를 실행하면 키와 값이 출력된다. 블록에 포함된 모든 레코드에 대해 맵 함수가 호출되면서 해당 태스크가 실행된 장비에 로컬 임시 파일이 생성된다.
* 파티셔너는 키 값을 이용하여 맵 함수의 결과를 전달할 리듀서를 결정한다. 작업을 실행할 때 리듀서 개수를 미리 지정하기 때문에, 리듀서의 개수는 작업이 실행되는 시점에는 이미 정해져 있다. 파티셔너는 리듀서 개수와 파티셔닝 전략에 따라 맵 함수의 출력 데이터를 처리할 리듀서를 결정한다. 기본적으로 제공되는 파티셔닝 전략은 해시 알고리즘으로, ‘hash(키) % 리듀서 개수”다. 즉 동일한 키를 가지는 데이터는 모두 동일한 리듀서로 전달된다.
* 결국 맵 태스크에 생성되는 로컬 임시 파일은 기본적으로 ‘맵 태스크 id – 리듀서 태스크 id’라는 이름을 가진다.
* 맵 태스크에서 로컬 임시 파일 생성이 완료되면, 리듀서 태스크는 각 태스크 트래커로 맵 태스크의 임시 파일 전송을 요청한다. 태스크 트래커는 맵 태스크의 상태를 모니터링하고 있으면서, 태스크의 수행이 완료되면 임시로 생성된 파일을 리듀서 태스크로 전달한다.
* 모든 맵 태스크가 완료되면 리듀서 태스크에도 최대한 맵 태스크 개수만큼의 파일이 복사되어 있다.
* 모든 맵이 완료되면 리듀서 태스크에서는 복사된 파일을 병하하면서 키로 정렬한다. 정렬작업이 완료되면 결국 하나의 키에 대해 각 맵에서 출력된 값을 목록으로 만들어진다.
* 이렇게 정렬된 데이터가 사용자가 개발한 리듀스 함수로 전달되고, 리듀서 함수에서 출력된 결과는 기본적으로 하둡 파일 시스템에 저장된다. 기본적으로 리듀스 태스크마다 출력파일이 하나씩 생성된다.

<지역성 보장>

잡 트래커는 태스크 트래커가 작업할 수 있는 파일이 해당 장비에 있을 경우에 우선적으로 태스크를 할당한다. 따라서 비록 파일이 여러 노드에 분산되어 저장되어 있더라도, 실제로는 태스크가 자신의 로컬 장비에서 파일을 읽는 것과 유사한 성능을 보장할 수 있게 된다.

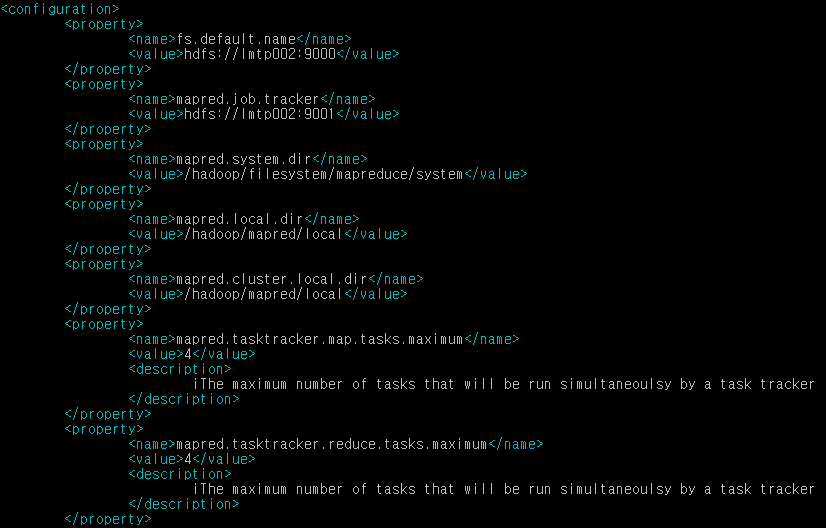
### 설치

맵리듀스는 하둡 파일 시스템과 동일한 패키지로 배포되므로, 앞의 하둡 파일 시스템을 이미 설치하였다면 맵리듀스 역시 설치가 완료된 상태다. 따라서 맵리듀스를 실행하기 위해서는 환경 설정과 관련된 파일ㅇ만 설정하면 된다.

#### 환경 설정

맵리듀스와 관련된 설정은 conf/mapred-site.xml에 설정한다. 기본값은 src/mapred/mapred-default.xml 파일에서 확인할 수 있다.

* mapred-site.xml



#### 실행과 관리

맵리듀스 시스템은 잡 트래커 서버에서 실행한다. 태스크 트래커는 conf/slaves 파일에 있는 서버 목록에 대해 원격 ssh 명령어를 수행하여 실행된다.

하둡 파일 시스템을 실행한다.

bin/start-mapred.sh

맵리듀스 시스템 또한 모니터링할 수 있는 웹 화면을 제공한다. 웹 화면은 <http://namenode_hostnam:50030> 으로 접속한다.

[그림. 맵리듀스 시스템 웹 모니터링 화면]



#### 작업 관리 명령

콘솔에서 맵리듀스 잡을 관리할 때 아래와 같은 명령어를 사용할 수 있다.

[[ 맵리듀스 잡 관리 명령어 추가할 것 ]]

## 맵리듀스 프로그래밍

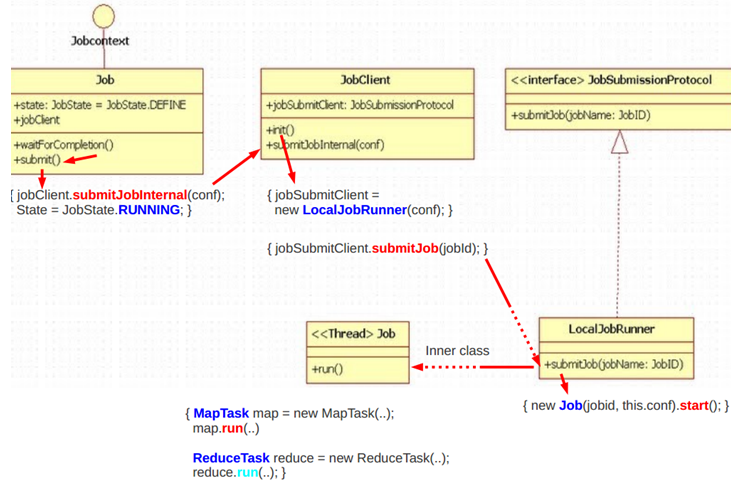
단어 세기 유스케이스를 바탕으로 맵리듀스 프로그래밍의 구성 요소를 좀 더 자세히 살펴보자.

### WordCount 예제

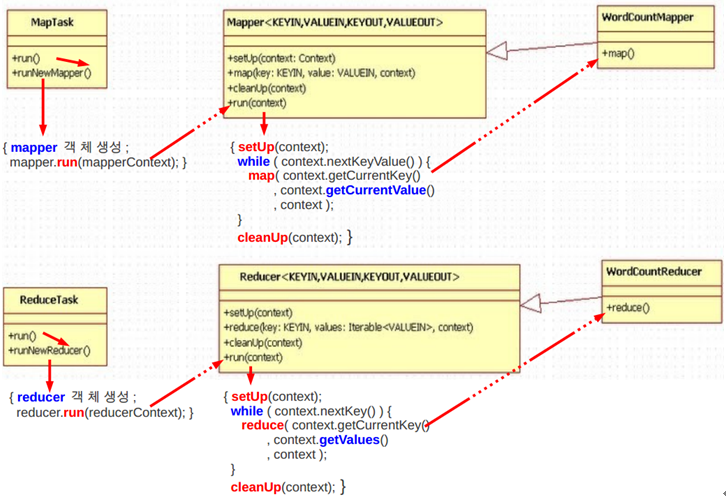
WordCount 예제와 관련된 클래스들을 살펴보자.

#### 클래스 다이어그램

맵리듀스 잡을 실행하는 단계에 관련된 클래스는 다음과 같다.



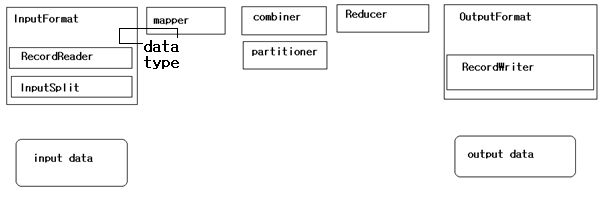
맵 함수와 리듀서 함수가 실행되는 단계와 관련된 클래스 다이어그램은 다음과 같다.



### 맵리듀스 프로그래밍 구성 요소

맵리듀스 구성요소는 다음과 같다.

[그림. 맵리듀스 구성 요소]

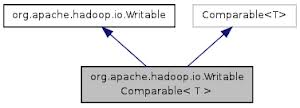


#### 데이터 타입

매퍼 또는 리듀서의 입/출력 값으로 키와 값의 쌍이 전달된다. 하지만 모든 자바 데이터 타입을 키와 값의 타입으로 사용할 수는 없다. 맵 리듀스 프로그램은 네트워크로 연결된 다수의 장비에서 실행되므로, 입/출력 데이터 타입은 반드시 직렬화/역직렬화가 가능해야 한다.

다시 말해 입/출력 데이터 타입은 반드시 Writable 인터페이스를 구현해야 한다. 또한 키는 정렬이 가능해야 하므로 WritableComparable<T> 인터페이스를 구현해야 한다.

[ 그림. WritableComparable<T> 인터페이스 상속 관계]



기본적으로 하둡은 WritableComparable<T>를 구현한 데이터 타입을 제공한다.

[표. 하둡 데이터 타입]

|  |  |
| --- | --- |
| 클래스 | 설명 |
| BooleanWritable |  |
| ByteWritable |  |
| DoubleWritable |  |
| FloatWritable |  |
| IntWritable |  |
| LongWritable |  |
| Text |  |
| NullWritable |  |

#### InputFormat

InputFormat은 2가지 역할을 수행한다. 먼저 하나의 잡이 몇 개의 맵 태스크로 분리 되어야 할지를 결정한다. InputFormat의 getSplits() 메서드는 InputSplit 배열을 반환하는데, 이때 반환되는 배열의 개수가 맵 태스크의 개수다. 예를 들어 하둡에서 기본적으로 제공하는 FileInputFormat 클래스의 getSplits() 메서드는 입력 데이터의 블록 개수(복제본 제외)만큼의 InputSplit 객체를 반환한다.

두번째로, 입력 데이터를 실제로 읽어서 맵 태스크의 map() 메서드로 전달하는 기능을 수행한다. 이 기능은 RecordReader 클래스가 담당하며, RecordReader 객체는 InputFormat 클래스의 createRecordReader() 메서드를 호출하여 생성할 수 있다. 예를 들어 FileInputFormat이 반환하는 RecordReader는 파일에서 하나의 라인을 읽어서 map() 메서드로 전달한다.

#### Mapper

RecordReader에 의해 전달된 각 레코드를 처리한다.

Mapper 클래스를 상속받는다.

#### Combiner

각 맵이 수행된 후 생성된 임시 로컬 데이터를 리듀서로 보내기 전에, 맵 태스크 내에서 리듀스 작업을 수행하는 클래스다.

맵리듀스 잡이 실행될 때 시간이 기장 오래 걸리는 부분은 맵 결과가 리듀스로 복사되서 정렬되는 과정이다. 컴바이너를 사용하면 맵의 출력 결과를 로컬에서 먼저 리듀스를 수행하므로, 리듀서로 전달하는 데이터의 야을 줄일 수 있으므로 맵리듀스 잡의 성능을 향상한느 효과가 있다.

일반적으로 컴바이너는 리듀서 클래스를 그대로 활용한다. 잡 드라이버에서 setCombiner() 메서드로 지정햐아여만 실행된다.

#### Partitioner

키를 이용하여 맵의 출력 데이터를 전달할 리듀스를 결정한다.

#### Reducer

매퍼에 의해 전달된 데이터를 처리한다.

#### OutputFormat

리듀서의 출력 데이터를 저장한다. 출력 데이터는 RecordWriter가 담당하며, RecordWriter 객체는 OutputFormat 클래스의 getRecordWriter() 메서드가 생성한다.

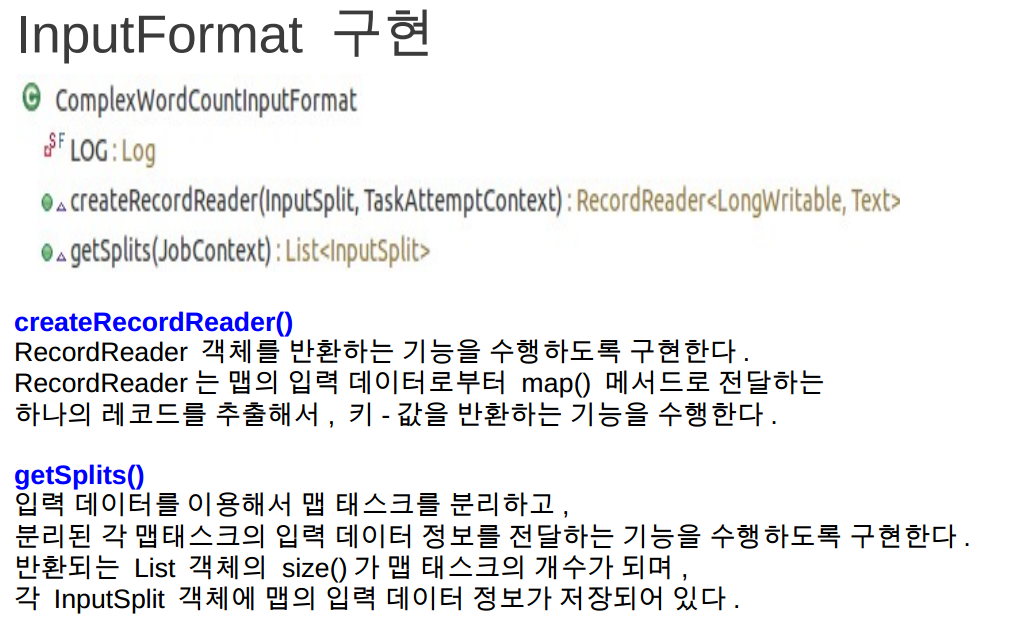
### 맵리듀스 프로그래밍 개발 절차

맵리듀스 어플리케이션을 아래의 절차를 따라서 개발한다.

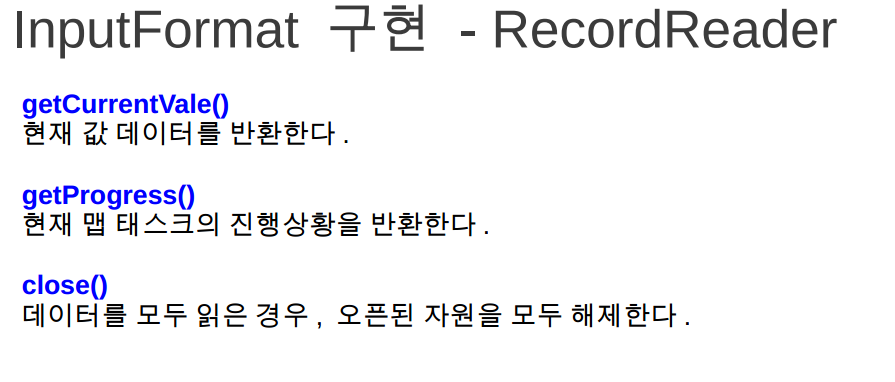
* 데이터 흐름 설계
* InputFormat/OutputFormat 구현
* Mapper 구현
* Partitioner 구현
* Reducer 구현
* Driver 구현

#### 데이터 흐름 설계

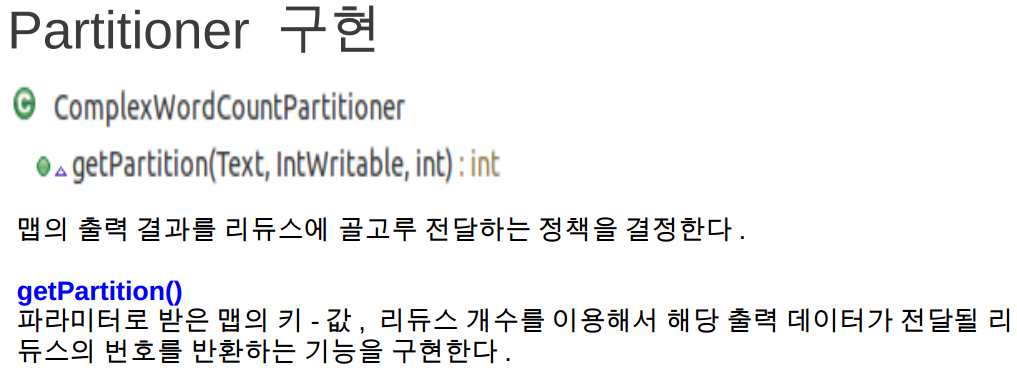
#### InputFormat 구현







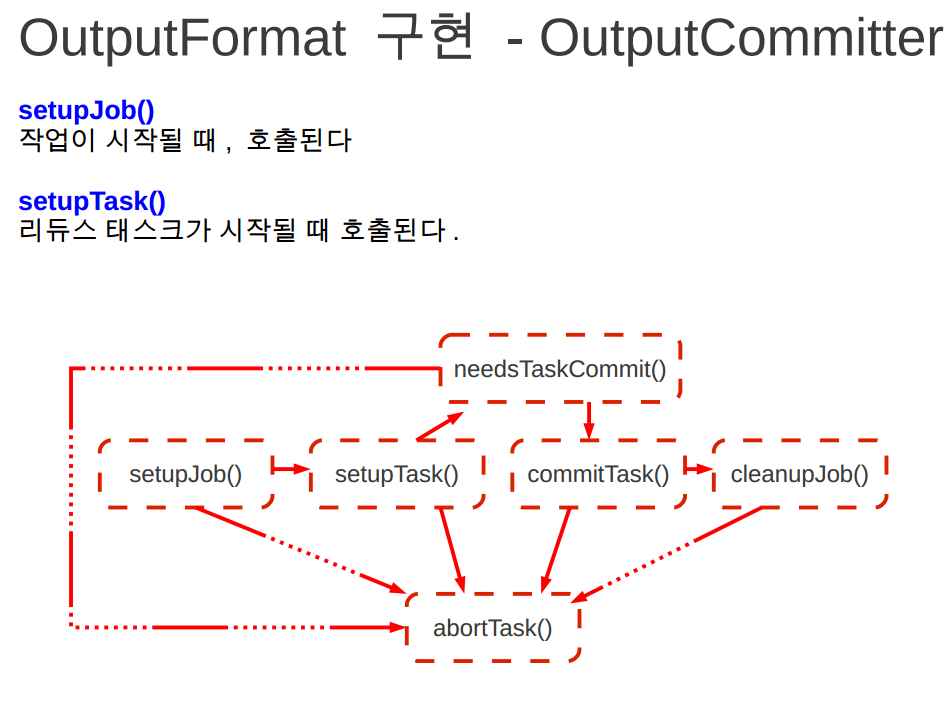
#### Partitioner 구현

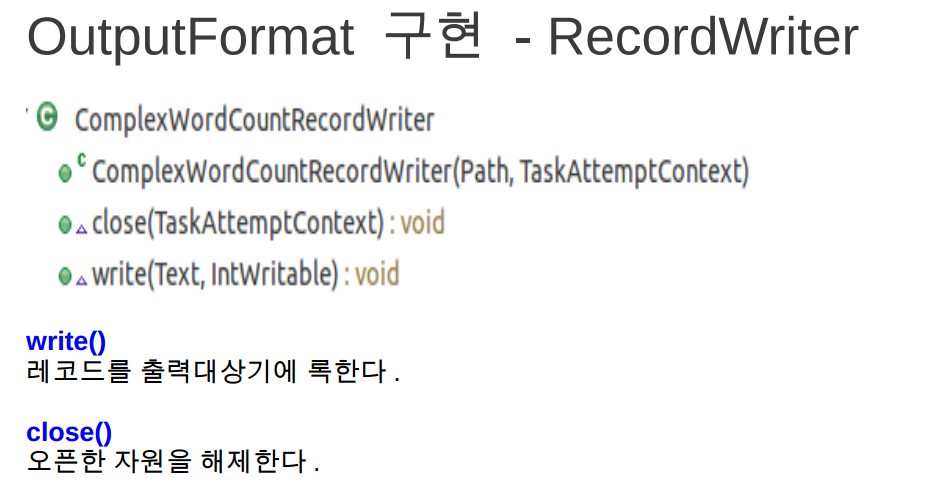


#### OutputFormat 구현







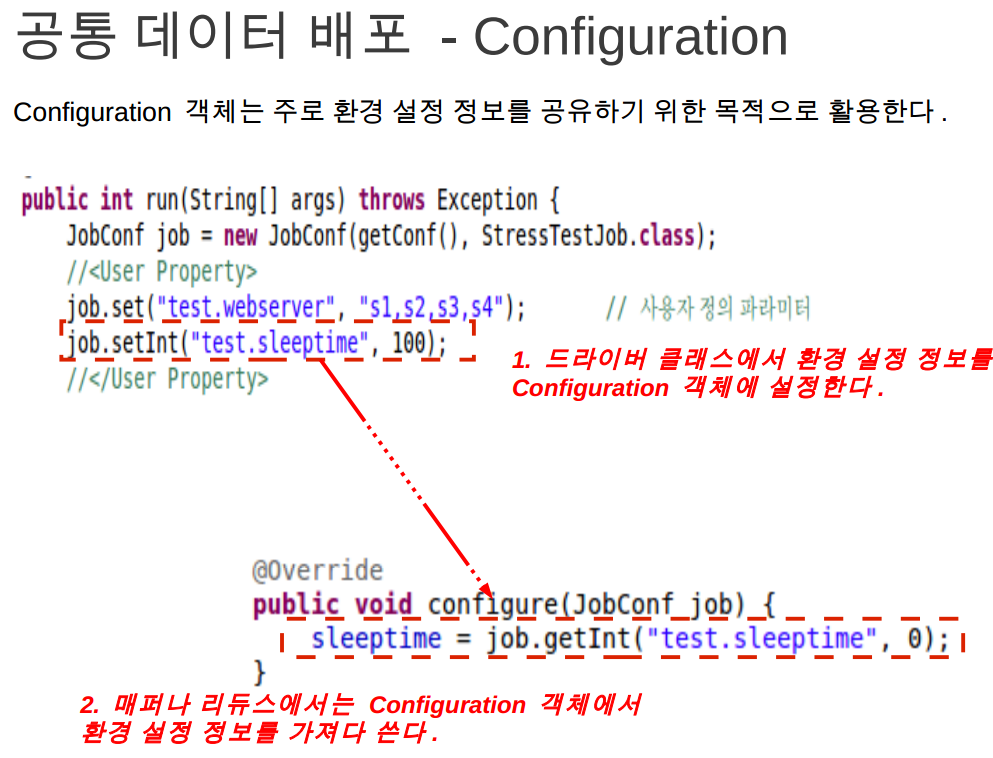


### 공통 데이터 배포

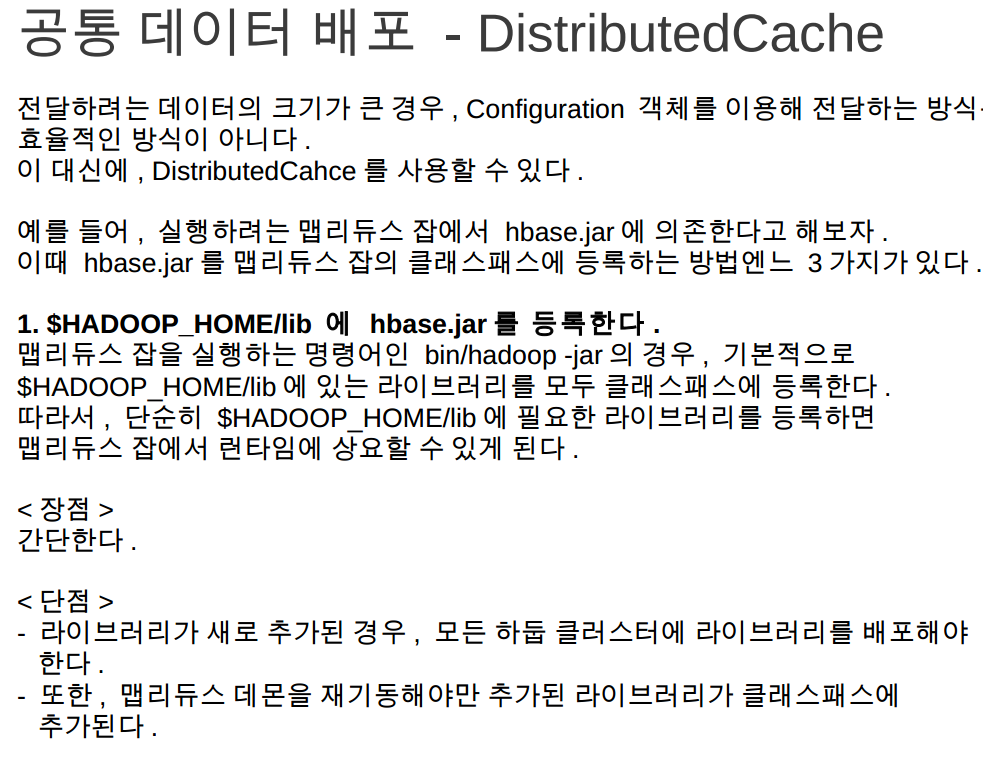
맵리듀스 프로그램은 분산된 서버에서 실행된다. 이처럼 분산 환경에서 프로그램을 실행할 때 어려운 일은 공통 데이터(코드 데이터), 외부 라이브러리, 환경 설정 값 등을 공유하고 배포하는 일이다.

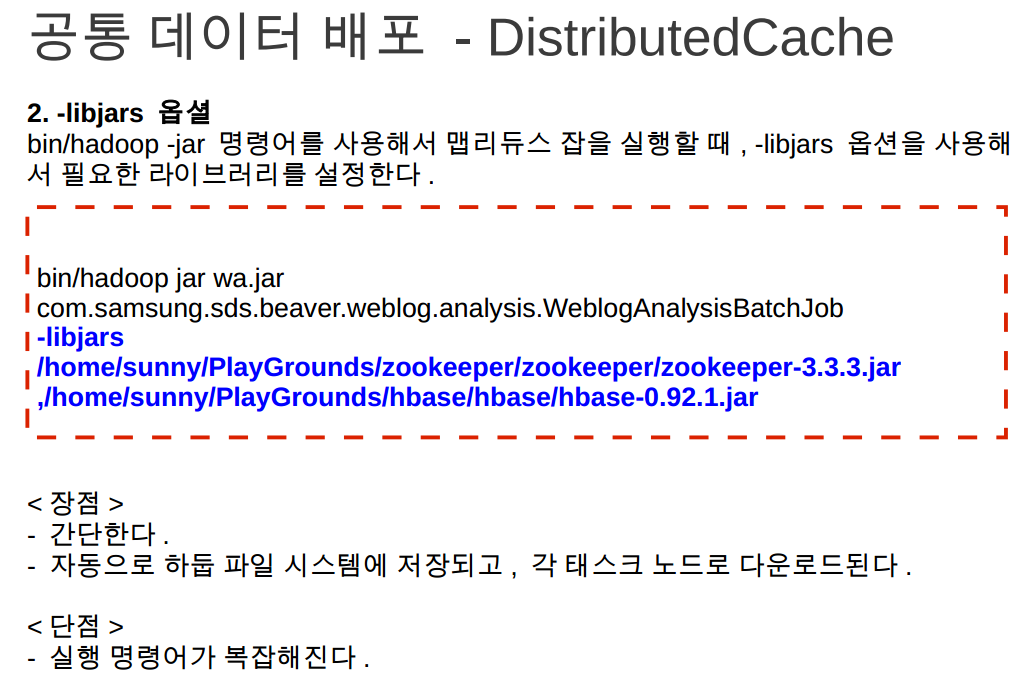
환경 설정 값 또는 간단한 코드 데이터를 공유하려면 Configuration 객체를 사용할 수 있다. 또한 외부 라이브러리, 또는 대량의 코드 데이터를 배포하려면 DistributedCache 등을 사용할 수 있다.

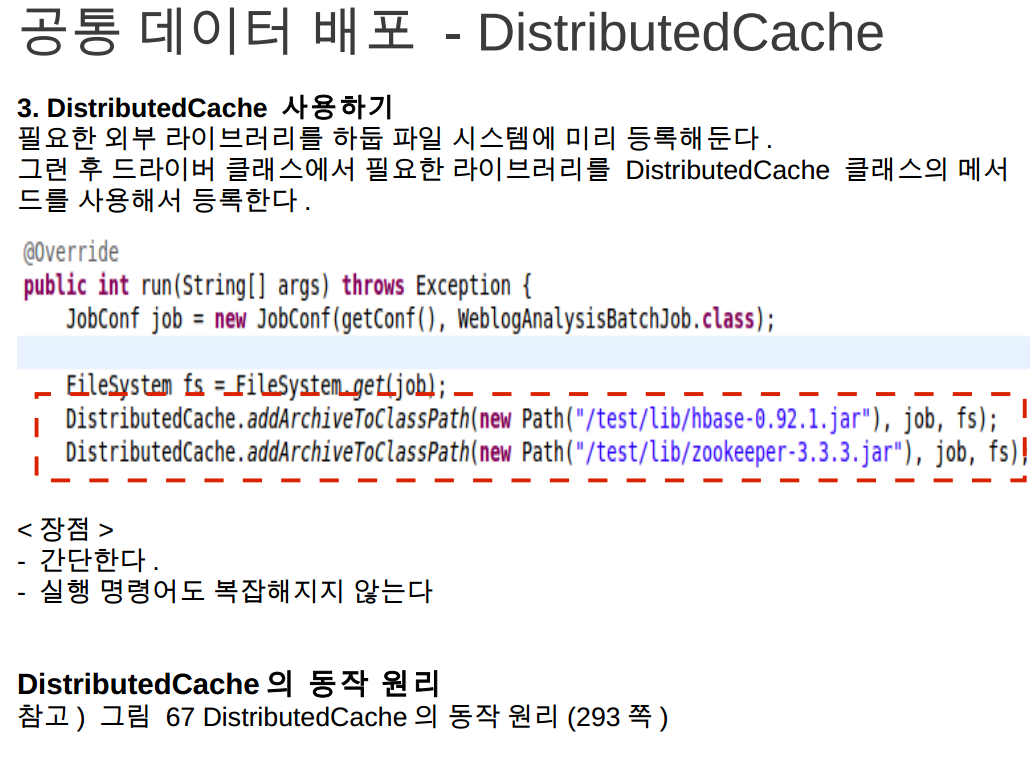
#### 환경 설정 값 공유하기



#### 외부 라이브러리 공유하기







### 정렬

[[ 조금 더 공부할 것 ]]

### 조인

[[ 조금 더 공부할 것 ]]

[[ 조인과 관련된 내용 추가할 것 ]]

#### 맵 사이드 조인

#### 리듀서 사이드 조인

## 하이브(Hive)

### 개요

#### 시스템 구성

### 설치

### 실행

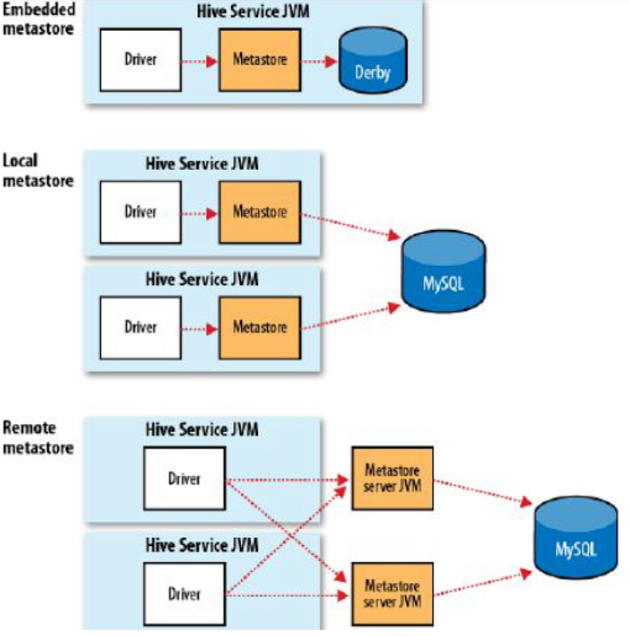
== Hive metastore(메타스토어)의 유형 및 설정 방식

**Hive metastore(메타스토어)란**

하이브는 테이블과 파티션과 관련된 메타정보를 모두 메타스토어에 저장한다.

**Hive 메타스토어의 유형**

하이브의 메타스토어 유형에는 임베디드 메타스토어(Embedded metastore), 로컬 메타스토어(Local metastore), 원격 메타스토어(Remote metastore) 세가지 유형이 있다.



* **임베디드 메타스토어(Embedded metastore)**  
  하이브를 설치하면 기본적으로 임베디드 메타스토어를 사용한다. 이 경우 메타스토어가 로컬 장비에 파일로 생성되므로 한번에 하나의 프로세스만 메타스토어에 접근할 수 있다. 따라서 실제 환경에서 사용해서는 안된다.
* **로컬 메타스토어(Local metastore)**  
  로컬 메타스토어의 경우 메타데이터가 모두 원격(또는 로컬)의 데이터베이스에 저장된다.
* **원격 메타스토어(Remote metastore)**  
  원격 메타스토어의 경우에도 메타데이터가 모두 원격(또는 로컬)의 데이터베이스에 저장된다. 하지만 로컬 메타스토어와는 달리 메타스토어를 서비스하는 별도의 서버가 기동되며, 클라이언트는 데이터베이스에 집접 쿼리문을 날리는 대신 메타스토어 서버의 중개를 받게 된다. 이때 클라이언트와 메타스토어 서버는 thrift 통신을 사용한다.

**Hive 메타스토어의 설정 파라미터**

하이브의 메타스토어를 설정하는 파라미터는 다음과 같다.

|  |  |
| --- | --- |
| 파라미터 | 설명 |
| javax.jdo.option.ConnectionURL | 메타데이터를 저장하는 DB의 접속 정보 |
| javax.jdo.option.ConnectionDriverName | DB에 연결할 때 사용할 JDBC 드라이버명 |
| hive.metastore.uris | 클라이언트에서 메타스토어 서버와 통신하는 URI |
| hive.metastore.local | 임베디드 또는 로컬인 경우 true, 원격인 경우 false |
| hive.metastore.warehouse.dir | 하이브 테이블이 저장되는 HDFS 상의 경로 |

**Hive 메타스토어의 유형별 설정 방법**

하이브의 메타스토어 유형별로 아래와 같이 설정한다. 기본적으로 아무 설정도 하지 않는다면 임베디드 메타스토어 방식을 사용한다. 아래의 파라미터를 설정하려면 hive-site.xml 파일을 만든 후, 기본 값을 변경할 파라미터만 설정한다.

* **임베디드 메타스토어(Embedded metastore)**

|  |  |
| --- | --- |
| 파라미터 | 설정 값 |
| javax.jdo.option.ConnectionURL | jdbc:derby:;databaseName=../build/test/junit\_metastore\_db  ;create=true |
| javax.jdo.option.ConnectionDriverName | org.apache.derby.jdbc.EmbeddedDriver |
| hive.metastore.local | true |
| hive.metastore.warehouse.dir | /user/hive/warehouse |

* **로컬 메타스토어(Local metastore)**

|  |  |
| --- | --- |
| 파라미터 | 설명 |
| javax.jdo.option.ConnectionURL | jdbc:mysql://<host name>/<database name>?  createDatabaseIfNotExist=true |
| javax.jdo.option.ConnectionDriverName | com.mysql.jdbc.Driver |
| javax.jdo.option.ConnectionUserName | <user name> |
| javax.jdo.option.ConnectionPassword | <password> |
| hive.metastore.local | true |
| hive.metastore.warehouse.dir | /user/hive/warehouse |

* **원격 메타스토어(Remote metastore)**  
  서버는 아래와 같이 설정한다.

|  |  |
| --- | --- |
| 파라미터 | 설명 |
| javax.jdo.option.ConnectionURL | jdbc:mysql://<host name>/<database name>?  createDatabaseIfNotExist=true |
| javax.jdo.option.ConnectionDriverName | com.mysql.jdbc.Driver |
| javax.jdo.option.ConnectionUserName | <user name> |
| javax.jdo.option.ConnectionPassword | <password> |
| hive.metastore.warehouse.dir | /user/hive/warehouse |

* 클라이언트는 아래와 같이 설정한다.

|  |  |
| --- | --- |
| 파라미터 | 설명 |
| hive.metastore.uris | thrift://<host\_name>:<port> |
| hive.metastore.local | false |
| hive.metastore.warehouse.dir | /user/hive/warehouse |

**[참고자료]**

* [Apache Hive Admin Manual : MetastoreAdmin](https://cwiki.apache.org/Hive/adminmanual-metastoreadmin.html)

== Hive Web Interface 링크

**HWI(Hive Web Interface)**

* https://cwiki.apache.org/Hive/hivewebinterface.html
* http://sql.dzone.com/articles/hadoop-hive-web-interface
* http://stackoverflow.com/questions/2571198/hadoop-hive-web-interface-options

**Cloudera HUE(Hadoop User Experience) & Beeswax**

* http://www.slideshare.net/cwsteinbach/cloudera-huebeeswax
* https://ccp.cloudera.com/display/CDHDOC/Beeswax#Beeswax-BeeswaxandHiveInstallationandConfiguration
* https://ccp.cloudera.com/display/CDHDOC/Hue+Installation

**Templeton**

* http://geekdani.wordpress.com/2012/07/18/hcatalog-templeton/
* https://speakerdeck.com/dgkim84/hcatalog-templeton

== Hive 설치하기 - 원격 메타스토어 방식

**필요 조건**

* 하둡 설치
* 하이브(0.50 버전 이상) 임베디드 모드(default)로 설치
* MySQL 설치

여기에서는 하나의 장비에 서버와 클라이언트를 모두 설치한다.

**서버 설정하기**

먼저 MySQL JDBC 커넥터 라이브러리를 $HIVE\_HOME/lib에 추가해야 한다.

hive-site.xml을 아래와 같이 설정한다.

<property>

  <name>**javax.jdo.option.ConnectionURL**</name>

  <value>jdbc:mysql://localhost:3306/hive?createDatabaseIfNotExist=true</value>

  <description>JDBC connect string for a JDBC metastore</description>

</property>

<property>

  <name>**javax.jdo.option.ConnectionDriverName**</name>

  <value>com.mysql.jdbc.Driver</value>

  <description>Driver class name for a JDBC metastore</description>

</property>

<property>

  <name>**javax.jdo.option.ConnectionUserName**</name>

  <value>root</value>

  <description>username to use against metastore database</description>

</property>

<property>

  <name>**javax.jdo.option.ConnectionPassword**</name>

  <value>root</value>

  <description>password to use against metastore database</description>

</property>

<property>

  <name>**hive.metastore.warehouse.dir**</name>

  <value>/user/hive/warehouse</value>

  <description>location of default database for the warehouse</description>

</property>

메타스토어 서비스를 기동한다. 메타스토어 서비스는 기본적으로 9083 포트로 기동된다. 만약 포트를 바꾸고 싶다면 -p 옵션을 사용한다. 자세한 옵션은 bin/hive --service metastore --help 명령어로 확인한다.

$ bin/hive --service metastore

[Ctrl - Z]

$ bg

**클라이언트 설정하기**

hive-site.xml을 아래와 같이 설정한다.

클라이언트를 원격에서 접속하는 경우라면, 클라이언트에도 MySQL JDBC 커넥터 라이브러리를 $HIVE\_HOME/lib에 추가해야 한다.

<property>

  <name>**hive.metastore.warehouse.dir**</name>

  <value>/user/hive/warehouse</value>

  <description>location of default database for the warehouse</description>

</property>

<property>

  <name>**hive.metastore.uris**</name>

  <value>thrift://localhost:**9083**</value>  
  <description>hive metastore service uri</description>

</property>

<property>

  <name>**hive.metastore.local**</name>

  <value>false</value>

  <description>controls whether to connect to remove metastore server or open a new metastore server in Hive Client JVM</description>

</property>

클라이언트 설정이 완료되면 이제 메타스토어 서비스에 접속해보자.

$ bin/hive

hive> show tables;

만약 show tables 명령어 실행시 아래와 같은 에러가 났다면,

FAILED: Error in metadata: MetaException(message:**Could not connect to meta store using any of the URIs provided**)

FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask

하이브 메타스토어 서비스 접속 정보(hive.metastore.uris)가 잘못된 것이므로 확인이 필요하다.

**[참고자료]**

* [Apache Hive Admin Manual : Meatastore Admin](https://cwiki.apache.org/Hive/adminmanual-metastoreadmin.html)

== RegexSerDe를 사용해서 Hive에 데이터 임포트하기

Hive 테이블 생성하기

CREATE TABLE web\_samsungcom\_6\_sec\_monthly (  
cid    STRING,  
sip    STRING,  
seq    STRING,  
client\_ip    STRING,  
log\_date    STRING,  
request    STRING,  
status    STRING,  
content\_length    STRING,  
headers\_x\_forwarded\_for    STRING )  
COMMENT ''  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

;

CREATE TABLE web\_samsungcom\_6\_sec\_monthly (  
client\_ip    STRING,  
log\_date    STRING,  
request    STRING,  
status    STRING,  
content\_length    STRING,  
headers\_x\_forwarded\_for    STRING )  
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'  
WITH SERDEPROPERTIES (  
'input.regex' = '([^\\s]+) - - \\[([^\\"]+)\\] \\"(.\*)\\" ([\\d]+) ([\\d]+) (.+)',  
'output.format.string' = '%1$s %2$s %3$s %4$s %5$s %6$s')  
;

LOAD DATA LOCAL INPATH "../data\_song/sec\_log\_10\_20/\*" INTO TABLE web\_samsungcom\_6\_sec\_monthly;

select \* from web\_samsungcom\_6\_sec\_monthly limit 2;

Caused by: org.apache.hadoop.hive.ql.metadata.HiveException: java.lang.ClassNotFoundException: org.apache.hadoop.hive.contrib.serde2.RegexSerDe

at org.apache.hadoop.hive.ql.exec.MapOperator.setChildren(MapOperator.java:406)

at org.apache.hadoop.hive.ql.exec.ExecMapper.configure(ExecMapper.java:90)  
클래스는 어떻게 추가하나?

create table web\_samsungcom\_sec\_monthly

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

as select 'SAMSUNG' domain, '33.33.108.115' aip, '1' seq, client\_ip, log\_date, request, status, content\_length, headers\_x\_forwarded\_for

from ( select \* from web\_samsungcom\_6\_sec\_monthly ) t1

where client\_ip is not null

hive-site.xml  
<property>

<name>hive.aux.jars.path</name>

<value>hdfs://lmms001:9000/lib/mr/hive\_contrib.jar</value>

<description>These JAR file are available to all users for all jobs</description>

</property>

## 피그(Pig)

### 개요

#### 시스템 구성

### 설치

### 실행

## HBase

### NoSQL 개요

관계형 데이터베이스는 ACID를 만족한다.  
[[ 실전 업무 설계 자료 참조 할 것]]

* Atomicity
* Consistency
* Isolation
* Durability

최근의 서비스는 엄격한 데이터 정합성보다는 확장성, 고가용성 등을 필요로 한다. 이러한 속성은 흔히 BASE라고 부른다.

* Basicaly Available  
  데이터는 언제라도 접근 가능해야 한다.
* Soft state  
  특정 시점에서는 데이터의 일관성이 보장되지 않을 수도 있다.
* Enventually consistent  
  일정 시간이 지나면 데이터의 일관성이 결국에는 만족된다.

이러한 데이터베이스를 NoSQL 데이터베이스라고 부른다.

#### RDBMS의 한계와 NoSQL의 등장

웹이 등장한 이후로 생성되는 데이터는 급격하게 증가하였으며, 모바일의 등장으로 증가추세가 급격해졌다. 이처럼 급격하게 증가하는 데이터를 저장하고 분석하는 일은 기존의 RDBMS로는 한계가 있다. 우선 RDBMS는 선형 확장(scale out)이 아닌 수직 확장(scale up)만이 가능하다. 뿐만 아니라 수직 확장에는 엄청난 비용이 든다. 하지만 이러한 서비스는 RDBMS가 제공하는 수준의 엄격한 정합성을 필요로하지 않는 경우가 많다.

* 조인 연산 불필요
* 정합성 보다는 가용성이 우선: 복제필요
* 데이터의 스키마가 정해지지 않음

#### CAP 이론

분산 환경에서 데이터 관리 시스템을 구성할 경우 CAP 이론이 있다.

* Consistency(정합성)  
  모든 클라이언트는 항상 동일한 데이터를 본다.
* Avaliability(가용성)  
  분산 시스템에서의 가용성이란, 네트워크 단절 상황에서도 노드 자체에 장애가 발생하지 않았다면 모든 요청에 대해 정해진 시간 내에 응답해야 한다.
* Partition Tolerance(단절내성)  
  네트워크가 단절된상태에서도 시스템의 속성(정합성 또는 가용성)을 유지해야 한다. 분산 환경에서는 서버들이 분산 배치되므로, 이 속성은 반드시 포함되어야 한다.

CAP 이론이란 브루어의 추측에서 시작되었으며, “적절한 응답 시간 내에 3가지 속성을 모두 만족시키는 분산 시스템을 구성할 수는 없다”이다.

[[ 그림. CAP 이론 삼각형 ]]

#### NoSQL의 특징과 분류

NoSQL 데이터베이스를 완전히 정의할 수는 없지만 일반적으로 다음과 같은 특징을 가지는 데이터베이스를 NoSQL이라고 부른다.

* 관계형 데이터 모델이 아닌 키-값 또는 키-값을 응용한 데이터 모델
* 안정적이 고가의 하드웨어가 아닌 다수의 값싼 하드웨어를 이용
* 데이터는 분산된 노드에 파티션, 복제되어 저장
* 데이터의 정합성에 대한 요구사항보다는 단절내성에 대한 요구사항이 더 중요
* 2단계 커밋 수준의 트랜잭션보단느 정족수 기반의 트랜잭션을 선호

NoSQL은 사용하는 데이터 모델에 따라 분류하거나, 분산하는 방식에 따라 분류할 수 있다.

<데이터 모델에 따른 분류>

* 키-값 모델

키와 바아너리 타입의 값을 저장하는 구조

키를 이용해서만 데이터 조회가 가능  
예) memcached, Dynamo, Redis

* 컬럼 모델

테이블, 컬럼과 같은 스키마가 존재하며, 데이터가 컬럼에 저장되는 구조  
예) Google Bigtable, Hbase, Cassandra

* 문서 모델

문서가 데이터의 저장 단위

하나의 문서 내에 여러 개의 필드와 필드에 대응하는 값이 있는 구조  
MongoDB

* 그래프 모델

노드와 에지로 구성된 그래프를 저장하며, 쉽게 탐색할 수 있는 구조

Neo4j

<분산방식에 따른 구조>

* 중앙 집중형  
  데이터의 배치 정보가 중앙의 데이터 서버 또는 마스터 서버에 저장되므로, 클라이언트가 데이터 연산을 하기 위해서는 메타데이터 또는 서버를 경유해 실제 데이터를 처리할 서버로 접속하는 방식  
  데이터 배치 정보를 중앙에서 관리하므로 관리가 용이

데이터를 관리하는 중앙 서버에 장애가 발생하면 전체 데이터에 접근할 수 없다.

예) Google Bigtable, HBase, MongoDB

* 분산형

별도의 메타 정보가 없으며, 해시 함수 등을 이용하여 특정 키를 서비스한느 서버를 찾는 방식

메타데이터를 관리하는 중앙 서버가 없으므로, 특정 서버에 장애가 나더라나도 해당 서버에서 서비스하는 키그룹만 서비스가 되지 않으며 나머지 서버는 정상적을 서비스가 가능. 즉 장애가 국소적임

#### NoSQL 선택 기준

현재 다양한 종류의 NoSQL이 존재하므로, 자신의 도메인과 요구사항에 가장 적합한 NoSQL을 선택해야 한다. NoSQL을 선택할 때 도움이 되는 기준은 다음과 같다.

* 데이터 복제 여부

데이터를 복제하여, 특정 노드에 장애가 나더라도 데이터 유실이 일어나지 않는가

* 특정 서버 장애시 데이터 연산 수행 가능 여부

특정 노드에 장애가 발생하더라도, 나머지 서버에서는 서비스가 가능한가

* 데이터 모델  
  요구사항에 가장 적합한 데이터 모델은 무엇인가
* 데이터 규모

데이터 규모가 선형적으로 확장 가능한가

* 관리의 편의성

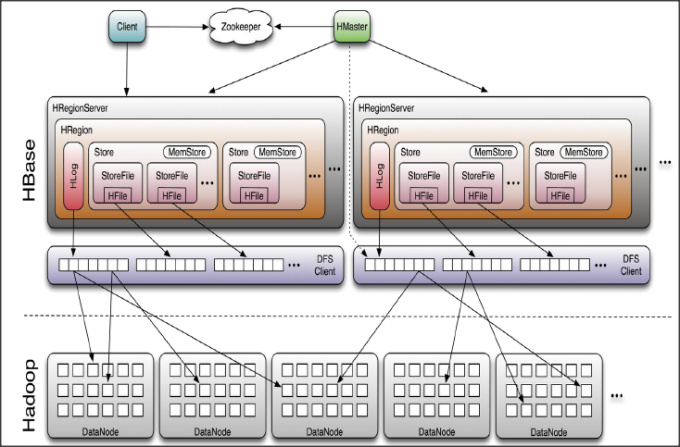
새로운 서버를 추가하고 쉽게 제거할 수 있는가

### HBase 소개와 구조

#### 시스템 구성

HBase는 마스트 서버, 리젼 서버로 구성된다. 또한 노드 관리를 위해 내부적으로 주키퍼를 사용한다.

[그림. HBase 시스템 구성]



마스터 서버는 테이블 생성, 삭제 등을 작업을 한다. 실제 데이터를 서비스하는 서버는 리젼 서버다. 하나의 리젼서버는 여러 개의 리젼을 관리한다.

리젼은 메모리 기반과 디스크 기반의 저장솔르 가진다. 메모리에 있는 데이터는 서버 중지시 유실을 방지하기 위해 HLog 객체가 관리하며, 실제 파일은 하둡 파일 시스템에 저장된다 하둡 파일 시스템에 저장된 파일은 불변 파일로, 한번 생성된 후 저장된 파일을 추가하거나 변경할 수 없다.

따라서 HBase의 커밋 로그는 변경 연산을 수행한 즉시 파일 시스템에 저장되는 것이 아니라, 주기적으로 저장된다. 따라서 커밋 로그가 저장되기 이전에 노드에 장애가 발생한 경우, 데이터 유실이 발생할 수 있다.

#### 데이터 모델

### 설치

#### 설치 요구 사항

#### 다운로드

#### 환경 설정

### 실행

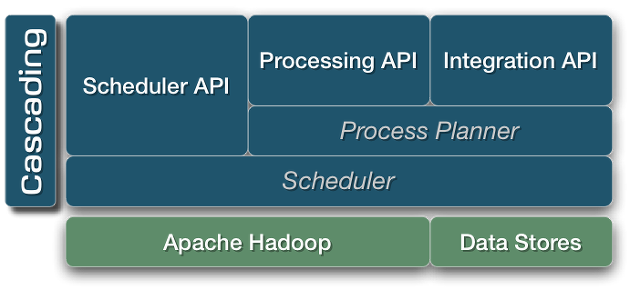
## 캐스캐이딩(Cascading)

== [번역] 캐스캐이딩(Cascading) 소개

원문 : http://www.cascading.org/about/

**빅 데이터 어플리케이션 개발**

캐스캐이딩(Cascading)은 평범한 개발자라도 풍부한 데이터 분석 및 관리(rich Data Analytics and Data Management) 어플리케이션을 쉽고 빠르게 개발할 수 있게 돕는 자바 기반의 어플리케이션 프레임워크다. 캐스캐이딩을 이용하여 만든 어플리케이션은 다양한 컴퓨팅 플랫폼에 배포하여 사용할 수 있다. 캐스캐이딩은 하둡 1.0과 호환되는 버전이라면 매끄럽게 동작한다.



**데이터 처리 API(Data Processing API)**

캐스캐이딩에서 가장 중요한 요소는 복잡한 데이터 흐름을 정의하고 데이터 중심의 정교한 프레임워크를 생성할 수 있는 풍부한 자바 API다. 이러한 프레임워크는 메이븐과 호환되는 라이브러리일수도 있고, 또는 스크립트 형태의 도메인 특화 언어(DSL)일 수도 있다.

**데이터 통합 API(Data Integration API)**

캐스캐이딩을 사용하면 통합과 관련된 복잡한 문제를 다루기 전에 먼저 기능을 만들어 테스트할 수 있다. 따라서 통합과 관련된 부분을 운영중인 데이터 흐름에 포함하기 전에 먼저 해당 부분을 개발해서 테스트할 수 있다.

**스케쥴러 API(Process Scheduler API)**

[riffle](https://github.com/cwensel/riffle) 라이프싸이클 어노테이션과 결합된 프로세스 스케쥴러를 사용하면, 어플리케이션이 어떤 써드 파티 라이브러리를 사용했더라도 캐스캐이딩에서는 단위 작업(unit of work)을 스케쥴할 수 있다.

**엔터프라이즈 개발**

캐스캐이딩은 어떤 엔터프라이즈 자바 개발 환경이라도 사용할 수 있도록 만들어졌다. 캐스캐이딩은 "데이터 처리"와 "데이터 통합"을 확실히 분리하였고, 또한 자바 API와 JUnit 테스팅 프레임워크도 분명하게 구분한다. 따라서 캐스캐이딩을 사용하면 규모에 상관없이 쉽게 테스트할 수 있다. 실제로 캐스캐이딩 코어 개발 팀에서도 CI 서버에서 1,500개의 테스트를 매일 실행하며, 테스트가 완료된 라이브러리는 모두 캐스캐이딩의 공개 메이븐 레파지토리([conjars.org](http://conjars.org/))에 배포하고 있다.

**데이터 과학**

캐스캐이딩은 자바를 기반으로 하기 떄문에, JVM을 기반으로 하는 언어라면 어디든지 사용할 수 있다. 가장 잘 알려진 언어로는 스칼라(Scala), 클로저(Clojure), Jruby, Jython, 그리고 그루비(Groovy)가 있다. 캐스캐이딩 커뮤니티에서 스크립트와 쿼리문을 이들 언어 대부분에서 이미 만들었다. 따라서 분석과 기계 학습용 어플리케이션이 필요하거나, 운영에 필요하다면 쉽게 구할 수 있을 것이다. 더 자세한 내용은 [extension](http://www.cascading.org/extensions/) 페이지를 살펴보라.

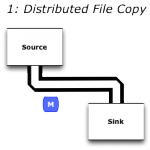
== [번역] 지금 당장 캐스캐이딩을 사용하기(1부)

원문 : http://www.cascading.org/2012/07/02/cascading-for-the-impatient-part-1/

오늘 배울 내용은 간단한 캐스캐이딩(Cascading) 2.0 앱을 만드는 방법이다. 목표는 간단하고 명확하다. 캐스캐이딩을 사용하여 가능한 단순한 어플리케이션을 최고의 실천법을 따라 만드는 것이다. 다시 말해 어떤 부가적인 요소도 가지지 않은, 하지만 효과적인 그런 코드 말이다.

<https://github.com/Cascading/Impatient/tree/master/part1>

위 링크에 열두어줄 정도로 작성된 자바 프로그램이 있다. 이 프로그램은 "A" 텍스트 파일의 라인을 "B" 파일로 복사한다. 이 프로그램에서는 1개의 매퍼 프로그램을 사용한다. 리듀서는 만들지 않는다. 이 프로그램에 대한 개념적인 다이어그램은 다음과 같다.



물론 이 작업은 리눅스의 cp와 같은 명령어를 사용해도 더 빨리 실행할 수 있다. 하지만 여기에서 설명할 예제는 단지 출발점일 뿐이다. 이 예제를 기반으로, 캐스캐이딩의 기능과 강점을 확인할 수 있는 새로운 코드들을 여기에 계속해서 추가해 나갈 것이다. 결국에는 일련의 문서에 포함된 키워드간의 상대적인 중요도(importance)에 대해 점수를 부여할 수 있는 TF-IDF 알고리즘을 구현한 맵리듀스 프로그램을 만들 것이다. 다시 말해 텍스트 마이닝에서 가장 기본적인 알고리즘을 구현할 것이다. 루씬(lucene)과 같이 문서를 인덱싱하는 프레임워크의 소스 코드에는 이러한 TF-IDF 알고리즘이 반드시 구현되어 있기 마련이다. 뿐만 아니라 확장 가능하면서도 견고한 맵리듀스 앱을 개발할 수 있도록, 캐스캐이딩에서 지원하는 TDD 기능을 활용하는 방법도 설명하고자 한다.

*\* 역주) TF-IDF : TF(Term Frequency)는 문서안에서 용어가 나오는 횟수, 또는 빈도를 의미한다. DF(Document Frequencey)는 용어가 나타나는 문서 개수를 의미하며, IDF(Inverse Document Frequency)는 DF의 역수다. 검색 관점에서 보자면, 검색하려는 용어가 많이 나타나는 문서가 우선순위가 높다. 즉 TF가 높은 문서가 우선순위가 높다. 반면 검색하려는 용어가 다수의 문서에서 나타난다면, 즉 DF가 높다면(IDF가 낮다면) 그 용어는 검색의 기준으로 변별력이 떨어진다고 볼 수 있다.*

**소스코드**

예제 코드를 [GitHub](https://github.com/Cascading/Impatient/tree/master/part1)에서 다운로로드한다. "지금 당장 캐스캐이딩을 사용하기" 시리즈의 전체 코드에 대한 클론(clone)을 가져와야 한다.

git clone git://github.com/Cascading/Impatient.git

이 예제에 대한 참고문서, 소스코드, 입력 데이터, 로그를 먼저 보고 싶다면, [gist](https://gist.github.com/2911686)를 살펴보라.

먼저 입력 데이터를 기술하기 위한 소스 탭(source tap)을 생성한다. 입력 데이터는 탭 문자를 기준으로 구분되는 TSV 형식이며, 헤더 행을 포함한다.

String inPath = args[ 0 ];  
Tap inTap = new Hfs( new TextDelimited( true, "\t" ), inPath );

다음으로 싱크 탭(sink tap)을 생성하여 출력 데이터를 기술한다. 출력 데이터도 마찬가지로 TSV 형식이다.

String outPath = args[ 1 ];  
Tap outTap = new Hfs( new TextDelimited( true, "\t" ), outPath );

그리고 나서 각 탭을 연결하기 위한 파이프(pipe)를 생성한다.

Pipe copyPipe = new Pipe( "copy" );

지금부터는 약간 재밌어진다. 약간의 배관작업을 해야 하니, 공구 벨트를 준비해야 할 것이다^^. 탭과 파이프를 플로우(flow)로 연결하자.

FlowDef flowDef = FlowDef.flowDef()  
                                      .addSource( copyPipe, inTap )  
                                      .addTailSink( copyPipe, outTap );

워크플로우(workflow) 개념이 캐스캐이딩에서 가장 핵심적인 요소다. 맵리듀스 job을 만들 때 매퍼와 리듀서 관점에서 생각하는 것과는 달리, 사람인 우리는 어플리케이션 자체에 대해 생각하기를 좋아한다. 실제의 경우 어플리케이션은 다수의 job이 여러 단계를 거쳐야 하는 경우가 많다. 이러한 단계들은 서로 연결되고 또한 의존성이 있다. 이러한 단계들은 방향성 비순환 그래프(DAG: Directed Acyclic Graph)로 흔히 표현된다. 캐스캐이딩에서는 FlowDef 객체를 사용하여 맵 리듀스 어플리케이션(즉 맵리듀스 job의 단계들)이 연결되는 방식을 정의할 수 있다.

이제 플로우까지 정의했으니, 마지막으로 해야할 일은 실행하는 코드를 기술하는 일이다.

flowConnector.connect( flowDef ).complete();

위의 소스 코드를 모두 main 메서드에 작성한 후, JAR 파일로 빌드한다. 문제 없이 빌드가 될 것이다. 이 예제에서 사용한 캐스캐이딩 API에 대해 좀더 자세히 알고 싶다면, 캐스캐이딩 2.0 사용자 가이드와 JavaDoc을 찾아보기 바란다.

**빌드하기**

The build for this example is based on using [Gradle](http://gradle.org/). The script is in build.gradle and to generate an [IntelliJ](https://www.jetbrains.com/idea/) project use:  
  
gradle ideaModule

To build the sample app from the command line use:  
  
gradle clean jar

What you should have at this point is a JAR file which is nearly ready to drop into your [Maven](https://maven.apache.org/) repo — almost. Actually, we provide a community jar repository for Cascading libraries and extensions at [http://conjars.org](http://conjars.org/)

**실행하기**

Before running this sample app, you’ll need to have a supported release of [Apache Hadoop](http://hadoop.apache.org/) installed. Here’s what was used to develop and test our example code:  
  
$ hadoop version  
Hadoop 1.0.3

Be sure to set your HADOOP\_HOME environment variable. Then clear the output directory (Apache Hadoop insists, if you’re running in standalone mode) and run the app:  
  
rm -rf output  
hadoop jar ./build/libs/impatient.jar data/rain.txt output/rain

Notice how those command line arguments align with args[] in the source. The file data/rain.txt gets copied, TSV row by TSV row. Output text gets stored in the partition file output/rain which you can then verify:  
  
more output/rain/part-00000

Again, here’s a [log file](https://gist.github.com/2911686) from our run of the sample app. If your run looks terribly different, something is probably not set up correctly. Drop us a line on the [cascading-user](https://groups.google.com/forum/?fromgroups#!forum/cascading-user) email forum. Plenty of experienced Cascading users are discussing **taps** and **pipes** and **flows** there, and eager to help. Or visit one of our user group meetings. [Coming up soon...]

Also, compare these other excellent implementations of the example apps here – by [Sujit Pal](http://sujitpal.blogspot.com/2012/08/scalding-for-impatient.html) in [Scalding](https://github.com/twitter/scalding/wiki) and by [Paul Lam](https://github.com/Quantisan/Impatient) in [Cascalog](https://github.com/nathanmarz/cascalog/wiki).

For those who are familiar with [Apache Pig](http://pig.apache.org/), we have included a comparable script:  
  
copyPipe = LOAD '$inPath' USING PigStorage('\t', 'tagsource');  
STORE copyPipe INTO '$outPath' using PigStorage('\t', 'tagsource');

To run that, use:  
  
rm -rf output  
pig -p inPath=./data/rain.txt -p outPath=./output/rain ./src/scripts/copy.pig

That’s it in a nutshell, our simplest app possible in Cascading. Not quite a “Hello World”, but more like a “Hi there, bus stop”. Or something. Stay tuned for the next installments of our *Cascading for the Impatient* series.

## 우지(oozie)

### 개요

#### 시스템 구성

### 설치

### 실행

## elasticsearch

### 개요

#### 시스템 구성

### 설치

### 실행

== 로그 파일에 대해 Elasticsearch 사용하기

원문 : [Using Elasticsearch for logs](http://www.elasticsearch.org/tutorials/2012/05/19/elasticsearch-for-logging.html)

By [Radu Gheorghe](http://www.twitter.com/) | 19 May 2012

로그를 저장하기 위해 Elasticsearch를 사용하고자 한다면, 이 문서를 통해서 Elasticsearch를 구성하는 방식에 대한 도움을 얻을 수 있다.

다수의 장비에서 로그를 수집하여 Elasticsearch에 저장하고자 한다면, 아래에서 하나를 선택해볼 수 있다.

* [**Graylog2**](http://graylog2.org/)   
  Graylog2를 중앙 서버에 설치하면, Graylog2가 알아서 Elasticsearch에 로그를 저장해준다. 그러면 깔끔한 인터페이스를 사용해서 저장된 로그를 검색할 수 있다.
* [**Logstash**](http://logstash.net/)  
  꽤 다양한 기능을 제공한다. 저장할 수 있는 로그의 종류(input)가 다양하며, 로그를 변환할 수 있는 방식(filter)도 많고, 변환된 로그를 저장할 수 있는 대상(output) 또한 다양하다. 무엇보다도 Elasticsearch에 직접 저장할 수 있는데, 이때 RabbitMQ의 [Elasticsearch river](http://www.elasticsearch.org/guide/reference/river/rabbitmq.html)를 사용할 수 있다.
* [**Apache Flume**](https://cwiki.apache.org/FLUME/)  
  플럼(Flume)을 사용하면 로그를 데이터소스(data source)로부터 수집한 후, 데코레이터(decoratro)를 사용하여 변환한 다음, 다양한 싱크(sink)를 이용해서 로그를 저장할 수 있다. 그중에서도 [elasticflume sink](https://github.com/Aconex/elasticflume)를 살펴보라.
* **rsyslog에 대한 omelasticsearch output 모듈**  
  어플리케이션 서버에서 rsyslog를 사용하여 로그를 Elasticsearch에 직접 저장하거나, 또는 중앙 서버에서 rsyslog를 사용하여 로그를 저장할 수도 있다. 또한 2가지 방식을 모두 활용할 수도 있다. 설정 방식에 대해서는 [rsyslog 위키](http://wiki.rsyslog.com/index.php/HOWTO:_rsyslog_%2B_elasticsearch)를 살펴보라.
* **직접 구현**  
  예를 들어 로그를 Elasticsearch에 저장하도록 커스텀 스크립트를 직접 만들어볼 수도 있다.

위의 방법 중 어느 방식을 사용하느냐에 따라 최적화 방식 또한 달라진다. 하지만 어느 방식을 사용하느냐에 관계 없이, 아래의 가이드는 도움이 될 것이다.

**메모리와 열린 파일 갯수**

서버를 Elasticsearch 전용으로만 사용한다면, 일반적으로 전체 메모리 중에서 절반정도를 Elasticsearch에 할당하는게 가장 좋다. 나머지 절반은 시스템 캐시를 위해 사용되는데, 이 점 또한 따라야 할 중요한 기준이다. 메모리를 할당하기 위해서는 **ES\_HEAP\_SIZE** 환경 변수를 설정한다. Elasticsearch를 기동하기 전에 적당한 값(예를 들면 2G)으로 설정한다. 설정 가능한 또다른 환경 변수는 **ES\_JAVA\_OPTS**로, 실행 스크립트(elasticsearch.in.sh 또는 elasticsearch.bat)로 전달된다. 여기에서 해당 프로세스에 할당할 최소 메모리와 최대 메모리량을 설정하는 **-Xms** 와 **-Xmx** 부분을 찾는다. 이 두 속성을 모두 ES\_HEAP\_SIZE와 동일하게 설정하는 것이 좋다.

또한 열린 파일 갯수를 Elasticsearch에 맞게 적당한 숫자로 할당해야 한다. 일반적으로 32000 또는 64000 정도로 할당하는게 좋다. 열린 파일 갯수를 설정하는 방법은 [여기](http://www.elasticsearch.org/tutorials/2011/04/06/too-many-open-files.html)를 참고하라.

**인덱스의 갯수**

로그를 인덱싱할 때 고려해볼 수 있는 사항은 로그를 모두 하나의 인덱스에 저장하고, 이후에 ttl 필드를 이용하여 오래된 로그가 자동으로 지워지도록 설정하는 방식이다. 하지만 이 방식에는 문제가 따르는데, 로그가 많은 경우 ttl 필드를 사용하면 추가적인 오버헤드가 든다는 점이다. 뿐만 아니라 거대한 하나의 인덱스를 최적화하는 작업은 상당히 오래 걸릴 뿐만 아니라, 처리하는데 소모되는 자원 또한 많다. 따라서 시간 순서대로 인덱스를 생성하는게 가장 좋다. 예를 들면 인덱스를 YYYY-MM-DD와 같은 형태로, 날짜 단위로 생성할 수 있다. 어느 수준의 날짜 단위를 사용하느냐는 로그를 얼마나 유지하느냐에 따라 달라진다. 만약 로그를 1주일가량만 유지한다면, 일단위로 인덱스를 생성하는게 좋다. 반면에 로그를 1년간 유지해야 한다면, 월단위로 인덱스를 생성하는게 가장 나아 보인다. 너무 많은 인덱스를 생성하면, 인덱스 모두에 대해 검색을 한꺼번에 실행하는 경우 너무 많은 오버헤드가 드므로, 인덱스가 많아지는 일은 좋은 선택이 아니다.

이처럼 시간 순서대로 인덱스를 저장하기로 마음먹었다면, 검색할 떄 관련된 인덱스로만 범위를 좁힐 수도 있다. 예를 들어 대다수의 검색이 최근의 로그에 대해서만 실행된다면, "빠른 검색" 기능을 UI에 추가하여 최근의 인덱스만 검색하게 만들 수도 있다.

**인덱스 삭제 및 최적화**

시간 순서대로 인덱스를 저장하는 경우, 오래된 로그를 삭제하는 일은 아래와 같이 간단하다.

$ curl -XDELETE 'http://localhost:9200/old-index-name/'

삭제 작업은 순식간에 처리된다(비슷비슷한 크기의 여러 파일을 삭제하는 정도다). cron 작업에 등록하여 밤에 실행되도록 설정할 수 있다.

사용률이 낮은 시간대에 [인덱스를 최적화](http://www.elasticsearch.org/guide/reference/api/admin-indices-optimize.html)하는 일도 효과가 있다. 인덱스를 최적화하면 검색 속도를 향상할 수 있다. 시간 순서대로 인덱스를 생성하는 경우에 특히 효과적인데, 오늘 이전의 인덱스는 변경될 일이 전혀 없기 때문이다(가장 최근의 인덱스를 제외하면). 따라서 오래된 인덱스에 대해서 단 한번만 최적화를 수행하면 된다.

$ curl -XPOST 'http://localhost:9200/old-index-name/\_optimize'

**샤드와 복제본**

elasticsearch.yml를 설정하거나 또는 Rest API를 사용하여 인덱스별로 설정하는 일도 가능하다. 자세한 설정 방법은 [여기](http://www.elasticsearch.org/guide/reference/setup/configuration.html)에서 찾아볼 수 있다.

가장 관심을 가져야 하는 속성은 샤드와 복제본의 갯수다. 기본적으로 각 인덱스는 5개의 샤드로 분할된다. 그리고 클러스터에 하나 이상의 노드가 있다면, 각 샤드는 하나의 복제본을 가지게 된다. 따라서 각 인덱스는 모두 10개의 샤드를 가지게 된다. 클러스터에 새 노드가 추가되면, 샤드는 균형을 맞추기 위해 자동으로 다른 노드로 이동된다. 따라서 기본 값으로 설정된 인덱스가 하나 있고 Elasticsearch 클러스터에 11개의 서버가 있다면, 1개의 서버는 어떤 데이터도 저장하지 않게 된다.

각 샤드는 루씬(Lucene) 인덱스이므로, 샤드의 크기가 작을수록 새로운 데이터를 Elasticsearch에 더 빨리 추가할 수 있다. 따라서 인덱스의 샤드 갯수를 늘리면, 데이터를 더 빨리 추가할 수 있다. 이때 주의할 점은 시간 순서대로 인덱스를 생성한다면, 로그는 오로지 "가장 최신의" 인덱스에만 추가되고, "이전의" 인덱스는 전혀 변경되지 않는다는 점이다. 주의해야하는 이유는 샤드가 많으면 많을수록, 저장 공간을 많이 차지할 뿐만 아니라 검색 속도가 늦어지기 때문이다. 따라서 가장 적당한 샤드 갯수를 선택해야 하며, 이는 데이터 추가/검색에 대한 요구사항 및 사용중인 하드웨어 스펙에 따라 달라진다.

반면에 복제본을 저장하면 하나 이상의 노드가 다운되더라도, 클러스터 자체는 정상적으로 운영할 수 있게 된다. 복제본이 많으면 많을수록, 클러스터가 정상적으로 운영하고자 할 때 필요로 하는 노드의 갯수는 적어도 된다. 또한 검색할 때 복제본도 사용되므로, 복제본이 많을수록 검색 속도가 빨라진다. 하지만 인덱스를 생성하는 시간은 늦어지는데 primary 샤드에 변경이 발생한 경우, 변경사항을 나머지 복제본에 모두 반영해야하기 때문이다.

**매핑. \_source과 \_all**

매핑이란 문서를 색인하고 저장하는 방법을 정의하는 일이다. 에를 들어 각 필드가 어떤 타입을 가질지를 정의할 수 있다. 매핑이란 문서를 색인하고 저장하는 방법을 정의하는 일이다. 에를 들어 각 필드가 어떤 타입을 가질지를 정의할 수 있다. syslog 메시지라면 문자열 타입으로 정의할 수 있고, 심각도(severity)라면 정수형으로 정의할 수 있다. 매핑을 직접 정의하고자 한다면 [여기](http://www.elasticsearch.org/guide/reference/api/admin-indices-put-mapping.html)를 살펴보라.

매핑은 적당한 기본값을 가지며, 각 필드의 타입은 문서가 인덱스에 처음 추가될 때 자동으로 탐지된다. 하지만 기본값을 그대로 활용하는 대신 일일이 설정하길 원할 때도 있다. 예를 들어 새로운 인덱스에 맨 처음 입력된 로그가 "message" 필드에 오로지 숫자 값만을 가진다면, 해당 필드는 long 타입으로 매핑되게 된다. 그런 후 이후에 들어오는 로그에서 99% 이상이 해당 필드에 문자열을 가진다면, Elasticsearch는 해당 로그는 인덱스에 추가하지 않고, 로그에서 "message" 필드에 해당하는 부분이 long 타입이 아니라고 로깅을 남기게 된다. 이러한 경우에는 "message" : {"type" : "string"}와 같이 매핑을 명시적으로 정의해야만 한다. 매핑을 명시적으로 등록하려면 [여기](http://www.elasticsearch.org/guide/reference/api/admin-indices-put-mapping.html)를 살펴보라.

특히 시간 순서대로 인덱스를 생성한다면, 설정 파일에 **인덱스 템플릿(index templatge)**을 생성하는 편이 낫다. 상세한 내용은 [여기](http://www.elasticsearch.org/guide/reference/api/admin-indices-templates.html)를 살펴보라. 매핑 뿐만 아니라,샤드의 갯수와 같이  인덱스와 관련된 나머지 속성들도 인덱스 템플릿에 정의할 수 있다.

매핑할 때 문서의 \_source에 대한 압축 여부도 설정할 수 있다. **\_source**는 실제 로그 라인으로, 압축 여부를 true로 설정하면 인덱스의 크기를 줄일 수 있다. 또한 설정 방식에 대해 성능이 향상될 수도 있다. 일반적으로 CPU 성능보다는 메모리 사이즈나 디스크 I/O에서 부하가 발생했을 때, source를 압축하면 더 빠른 속도를 어등ㄹ 수 있다. source 필드에 대한 더 자세한 내용은 [여기](http://www.elasticsearch.org/guide/reference/mapping/source-field.html)를 살펴보라.

기본적으로 모든 필드는 개별적으로 인덱싱되지만, 이 외에도 Elasticsearch는 모든 필드를 합쳐서 **\_all**이라고 부르는 새로운 필드에 인덱싱한다. 이렇게 하면 좋은 점은 특정 조건으로 \_all 필드에 대해 검색을 수행하면, 필드에 관계 없이 매칭되는 결과를 얻을 수 있다는 점이다. 단점은 인덱스를 만들 때 CPU를 추가적으로 사용하게 되며, 인덱스의 크기가 증가된다는 점이다. 따라서 \_all 필드를 사용하지 않을 계획이라면, 비활성화하는 편이 낫다. 또는 일부 필드만 \_all 필드에 추가되도록 정의할 수도 있다. 자세한 내용은 [여기](http://www.elasticsearch.org/guide/reference/mapping/all-field.html)를 살펴보라.

**리프레쉬 주기**

Elasticsearch는 문서가 인덱스된 후, 해당 문서를 검색할 때 찾을 수 있으려면 인덱스를 리프레쉬해야한다는 점에서 준실시간 처리를 한다고 볼 수 있다. 기본적으로 인덱스는 매초마다 비동기적으로 자동으로  리프레쉬된다.

리프레쉬 작업은 상당히 자원소모가 많은 작업이므로, 리프레쉬 주기를 늘린다면 인덱스에 문서를 추가하는 속도를 늘릴 수 있게 된다. 리프레쉬 주기를 어느 정도로 설정하지는 사용자의 요구사항에 따라 다르다.

원하는 리프뤠시 주기는 인덱스 템플릿에 저장할 수 있다. 또는 elasticsearch.yaml 설정 파일에 설정하거나, 인덱스의 설정을 Rest API를 사용해도 설정할 수 있다.

또는 리프레쉬 주기를 -1로 설정하여 리프레쉬 자업을 해제하고, [Rest API를 사용](http://www.elasticsearch.org/guide/reference/api/admin-indices-refresh.html)하여 리프레쉬 작업을 수동으로 할 수도 있다. 대량의 로그를 한꺼번에 추가하는 경우라면 이처럼 수작업으로 리프레쉬하는게 맞다. 하지만 일반적인 상황이라면 두가지 선택권이 있다. 벌크 작업으로 인덱스를 생성할 때마다 리프레쉬 작업을 하거나, 매번 검색을 하기 전에 리프레쉬를 할 수도 있다. 어느 방식을 선택하든, 각각의 작업을 처리하는데 시간이 더 걸리게 된다.

**쓰리프트**

일반적으로 Rest API는 HTTP 통신을 통해 이루어지지만 쓰리프트 통신을 사용할 수도 있으며, 쓰리프트를 사용하는 경우가 더 빠르다. 쓰리프트를 사용하려면 [transport-thrift plugin](https://github.com/elasticsearch/elasticsearch-transport-thrift)을 설치해야 하며, 클라이언트에서도 쓰리프트 통신이 가능해야 한다. 예를 들어 클라이언트가 [pyes Python client](https://github.com/aparo/pyes)를 사용한다면, HTTP 통신시 기본포트로 사용하는 9200번 대신에 9500번으로 단순히 포트를 바꿔서 접속하게되면, 쓰리프트로 통신하게 된다.

**비동기 복제**

일반적으로 인덱스 처리는 모든 샤드에서 문서에 대한 인덱싱 작업이 끝나야만 리턴된다. 이 대신에 [인덱스 API](http://www.elasticsearch.org/guide/reference/api/index_.html)를 사용하여 replication를 async로 설정하면, 복제가 백그라운드에서 실행되게 된다. 인덱스 작업을 할 떄 API를 직접 호출하는 경우에만 이처럼 설정할 수 있다. 만약 기존에 만들어진 클라이언트(pyes나 rsyslog의 omelasticsearch)를 사용하는 경우라면, 해당 클라이언트에 이 옵션을 지원해야만 설정할 수 있다.

**쿼리보다는 필터 사용하기**

일반적으로 로그를 인덱싱할 때, 점수보다는 날짜를 기준으로 정렬하기를 원할 것이다. 이러한 경우에는 점수는 거의 의미가 없다. 따라서 조건에 맞는 로그를 찾고자 한다면 쿼리보다는 필터를 사용하는 편이 낫다. 필터의 경우 점수 계산을 실행하지 않을 뿐더러, 그 결과가 자동으로 캐싱되기 때문이다. 더 자세한 내용은 [여기](http://www.elasticsearch.org/guide/reference/query-dsl/)를 살펴보라.

**벌크 인덱싱하기**

인덱싱할 때 [벌크 API](http://www.elasticsearch.org/guide/reference/api/bulk.html)를 사용하는 게 좋다. 각 로그를 매번 인덱싱하는 것에 비해 벌크로 인덱싱하는 것이 훨씬 빠르다. 이 때 아래 두가지 사항을 고려해야 한다.

* 적당한 벌크의 크기 : 벌크할 로그의 크기는 설정에 따라 다르다. 만약 처음 시도해 보는 경우라면, pyes에서는 400을 기본값으로 사용한다.
* 벌크에 대한 제한 시간을 설정 : 버프에 로그를 추가한 후, 벌크가 특정 크기를 넘어야만 인덱스에 추가되는 경우라면 벌크의 크기 뿐만 아니라 벌크 작업에 제한 시간을 설정해야 한다. 만약 제한 시간을 설정하지 않는다면, 로그가 천천히 입력되는 경우 버퍼에 로그가 추가되겠지만 Elasticsearch에 추가되려면 상당한 시간을 기다려야 하기 때문이다.

# 3부 운영하기

## 하둡 운영하기

### 실행과 관리

#### 데이터 노드 관리

데이터 노드는 네임 노드와 주기적으로 hearbeat 통신을 한다. 네임 노두는 일정시간 hearbeat 통신을 보내지 않은 데이터 노들르 장애가 났다고 판단하고, 해당 데이터 노드에 저장되어 있는 모든 블록에 대한 복제본을 다른 노드에 생성한다.

장애가 난 경우가 아니라 CPU 또는 메모리 증설, 레이드 컨트롤러 교체, 하드 디스크 교체 등의 이유로 서버를 중지해야할 경우가 가끔 있다. 이때 여러 노드를 동시에 중지하게 되면 블록 유실이 발생할 수도 있다. 또는 블록 유실이 발생하지 않더라도, 갑자기 너무 많은 서버를 중지하면 네임 노드가 복제 계수를 맞추기 위해 한꺼번에 블록 데이터를 복제하기 시작하면 네트워크 트랙피이 갑자기 증가하여, 네트워크에 연결된 나머지 서비스가 정상적으로 운영되지 않을 수도 있다.

따라서 중지하려는 데이터 노드에 저장되어 있는 모든 블록이 나머지 노드에 모두 복제되었는지 확인한 후, 다음으로 중지하려는 데이터 노드를 중지하는 방식으로 순차적으로 진행해야 한다. 하둡에서는 이러한 작업을 수행하기 위한 기능인 디커미션(decommission) 기능을 제공한다.

먼저 제거할 데이터 노드의 호스트명을 특정 파일에 설정한다. 예를 들어 datanode\_exclude 파일에 중지할 데이터 노드를 아래와 같이 기술한다.

# datanode\_exclude 파일

server04

server06

그리고 해당 파일을 hdfs-sites.xml에 등록한다.

<property>

<name>dfs.hosts.exclude</name>

<value>/usr/home/hadoop/hadoop/datanode\_exclude</value>

</property>

이제 데이터 노드를 제거하기 위해 아래의 명령어를 실행한다.

bin/hadoop dfsadmin –refreshNodes

이 명령어를 수행하면 하둡 파일 시스퉴 웹 UI의 Decommisionioning Nodes 링크에 해당 서버에 대해 디커미션 작업이 진행중이다라는 정볼르 확인할 수 있다.

#### 사용자 루트 패스

[[ 사용자 개념 추가할 것]]

### 네임노드 백업 및 HA 구성

세컨데리 네임노드는 네임노드가 설치되는 서버와 동일한 사양의 서버에 설치해야 한다. 특히 네임 노드와 동일한 용량의 메모리를 가져야 한다.

#### 백업 네임 노드

#### 네임노드 HA 구성하기

잡 스케쥴러, 레이드

== 하둡 아카이브 가이드 (Hadoop Archives Guide)

**하둡 아카이브 가이드 (Hadoop Archives Guide)**

원문: http://hadoop.apache.org/common/docs/current/hadoop\_archives.html

* 개요
* 아카이브를 생성하는 방법
* 아카이브에서 파일을 탐색하는 방법
* 아카비브 예제
  + 아카이브 생성하기
  + 파일 탐색하기
* 하둡 아카이브와 맵리듀스(MapReduce)

**개요**  
하둡 아카이브는 특별한 포맷의 아카이브다. 하둡 아카이브는 파일 시스템 디렉토리에 매핑된다. 하둡 아카이브는 반드시 \*.har 확장자를 가져야 한다. 하둡 아카이브 디렉토리에는 메타데이터(\_index와 \_masterindex라는 이름으로 된 파일)와 데이터 파일(part-\*)이 포함된다. \_index 파일에는 아카이브에 포함된 파일들의 이름과 해당 파일이 part 파일 어디에 있는지를 나타내는 위치가 기록되어 있다.  
  
  
**아카이브를 생성하는 방법**  
명령어는 다음과 같다.

hadoop archive -archiveName name -p <parent> <src>\* <dest>

-archiveName은 생성할 아카이브의 이름이다. 예를 들어 foo.har과 같이 이름 지을 수 있다. 아카이브 이름은 반드시 .har 확장자를 가져야 한다. parent 인자는 아카이브할 파일들의 상대 경로를 기술할 때 사용한다. 예를 들어 다음과 같이 기술할 수 있다.

-p /foo/bar a/b/c e/f/g

이때 /foo/bar가 상위 경로에 해당하고, a/b/c와 e/f/g가 상위 경로에 대한 상대 경로다. 이처럼 아카이브를 생성할 때 맵 리듀스가 잡을 사용한다. 따라서 이 명령어를 실행하려면 맵리듀스 클러스터를 필요로 한다. 자세한 설명은 다음 절에서 확인할 수 있다.  
  
단순히 디렉토리 하나( /foo/bar)를 아카이빙하는 경우라면, 다음과 같이 실행할 수 있다.

hadoop archive -archiveName zoo.har -p /foo/bar /outputdir

**아카이브에서 파일을 탐색하는 방법**  
아카이브 자체는 파일 시스템 레이어로 나타난다. 따라서 fs 쉘 명령어를 모두 아카이브에 대해서도 사용할 수 있다. 하지만 일반적인 하둡 파일 시스템의 URI와는 다른 URI를 사용한다. 또한 주의할 점은 아카이브는 불변이라는 점이다. 따라서 rename하거나 delete, 그리고 create  명령어를 실행하면 에러가 리턴된다. 하둡 아카이브에 대한 URI는 다음과 같다.

har://scheme-hostname:port/archivepath/fileinarchive

스킴을 기술하지 않은 경우, 기반 파일 시스템이라고 가정한다. 따라서 이 경우 URI는 다음과 같다.

har:///archivepath/fileinarchive

**아카이브 예제**  
**아카이브 생성하기**

hadoop archive -archiveName foo.har -p /user/hadoop dir1 dir2 /user/zoo

위의 예제의 경우 아카이브할 디렉토리의 상대 경로로 /user/hadoop를 지정한 후, 아카이브를 생성한다. 이때 /user/hadoop/dir1와 /user/hadoop/dir2 디렉토리가 파일 시스템의 /user/zoo/foo.har로 아카아빙 된다. 아카이빙을 하더라도 입력 파일은 삭제되지 않는다. 아카이브를 생성한 후 입력 파일을 삭제하고자 한다면(사용 중인 네임스페이스를 줄이기 위해), 입력 파일을 직접 삭제해야 한다.  
  
  
**파일 탐색하기**  
하둡 아카이브에서 파일을 탐색하는 일은 파일 시스템에서 ls를 실행하는 일만큼이나 간단하다. 위의 예제를 따라 /user/hadoop/dir1 and /user/hadoop/dir2 디렉토리르 아카이브 한 경우, 아래의 명령어를 실행해서 해당 아카이브에 있는 파일을 모두 확인할 수 있다.

hadoop dfs -lsr har:///user/zoo/foo.har/

-p 인자가 어떤 중요성을 가지는 이해하려면 앞에서 본 예제를 다시 한번 살펴봐야 한다. 하둡 아카이브에 대해 아래와 같이 ls (lsr이 아니라) 명령어를 실행하면,

hadoop dfs -ls har:///user/zoo/foo.har

그 결과는 다음과 같다.

har:///user/zoo/foo.har/dir1  
har:///user/zoo/foo.har/dir2

알다시피 위의 아카이브는 아래의 명령어를 실행해서 생성했다.

hadoop archive -archiveName foo.har***-p /user/hadoop*** dir1 dir2 /user/zoo

만약 이 명령어가 아니라 아래의 명령어로 바꾸어 실행해서 아카이빙 한후,

hadoop archive -archiveName foo.har ***-p /user/*** hadoop/dir1 hadoop/dir2 /user/zoo

해당 하둡 아카이브에 대해 아래와 같이 ls를 실행하면

hadoop dfs -ls har:///user/zoo/foo.har

아래와 같은 결과가 나온다.

har:///user/zoo/foo.har/hadoop/dir1  
har:///user/zoo/foo.har/hadoop/dir2

보는 바와 같이 아카이브된 파일은 /user/hadoop이 아니라 /user/를 기준으로 아카이브 된다.  
  
  
**하둡 아카이브와 맵리듀스(MapReduce)**  
맵리듀스에서 하둡 아카이브를 사용하는 일은 디폴트 파일 시스템이 아니라 다른 입력 파일 시스템을 사용하도록 기술하는 일만큼이나 간단하다. 예를 들어 HDS에 /user/zoo/foo.har로 하둡 아카이브를 저장했고, 해당 아카이브를 맵리듀스의 입력으로 사용하는 경우를 생각해보자. 이때 입력 디렉토리를 har:///user/zoo/foo.har로 설정하기만 하면 된다. 하둡 아카이브는 파일 시스템으로 나타나므로, 맵리듀스에서는 입력으로하둡 아카이브에 있는 입력 파일을 모두  사용할 수 있게 된다.

== WebHDFS 사용하기

HTTP 프로토콜을 사용하여 하둡에 접근할 수 있다.

WebHDFS를 사용하려면 hdfs-site.xml에 아래와 같이 설정해야 한다.

<property>

        <name>**dfs.webhdfs.enabled**</name>

        <value>true</value>

</property>

그리고 네임노드를 재기동한다.

WebHDFS는 기본적으로 50075 포트를 사용한다고 되어있지만, 사실 50070 포트를 사용해야 동작한다. 예를 들어 아래와 같이 디렉토리의 정보를 조회할 수 있다.

curl -i "http://localhost:50070/webhdfs/v1/tmp?op=GETFILESTATUS"

결과는 다음과 같다.

HTTP/1.1 200 OK

Content-Type: application/json

Transfer-Encoding: chunked

Server: Jetty(6.1.26)

{"FileStatus":{"accessTime":0,"blockSize":0,"group":"supergroup","length":0,"modificationTime":1358860869711,"owner":"socurites","pathSuffix":"","permission":"755","replication":0,"type":"DIRECTORY"}}

[참고자료]

* [Hadoop 1.0.x : WebHDFS 사용하기](http://geekdani.wordpress.com/2012/06/08/hadoop-1-0-x-webhdfs-%EC%82%AC%EC%9A%A9%ED%95%98%EA%B8%B0/)
* [WebHDFS REST API](http://hadoop.apache.org/docs/r1.0.4/webhdfs.html#FsURIvsHTTP_URL)

## 맵리듀스 성능향상하기

맵리듀스 잡은 여러 단계를 거쳐서 실행되며, 따라서 각 단계별로 성능을 향상할 수 있는 요소가 있다.

### 맵 태스크 튜닝

* mapred.map.max.attempts  
  맵 태스크 실패 후 재시도하는 횟수  
  기본값 : 4
* mapred.task.timeout  
  맵리듀스 태스크 수행 중에, 잡 트래커가 진행사항 정보를 해당 시간동안 받지 못한 경우 해당 태스크를 실패로 간주하고, 다른 서버에서 동일한 태스크를 수행한다.  
  기본값 ; 600000(10분)
* mapred.child.java.opts  
  .태스크 트래커에서 fork.하는 맵리듀스 태스크의 자바가상 머신 실행 옵션으로, 주로 힙메모리를 설정할 때 사용  
  기본값 : -Xmx200m
* mapred.map.task.speculative.execution  
  동일한 태스크를 여분의 태스크 트래커에서 동시에 실행할지 여부  
  특정 서버의 문제로 인해 태스크 하나가 늦게 실행이 완료되면, 맵리듀스 잡의 전체 실행시간은 해당 태스크에 의해 결정된다. 따라서 맵리듀스 잡이 특정 서버에 종속되지 않도록, 동일한 태스크를 여분의 태스크 트래커에서 동시에 실행한후 가정 먼저 끝난 태스크의 결과값을 사용할 수 있다.  
  기본값 : true
* mapred.compress.map.output  
  맵태스크의 출력 결과를 압축할 지 여부  
  맵리듀스 잡은 기본적으로 대용량의 데이터에 대해 동작하므로, 맵의 출력 결과 역시 큰 경우가 많다. 또한 맵의 출력 결과는 리듀서 태스크로 네트워크를 통해 전달되어야 하므로, 대다수의 맵리듀스 잡의 성능은 이러한 맵의 출력 결과를 리듀서로 보내는 단계에 의해 결정되는 경우가 많다. 따라서 맵의 출력 결과를 압축한다면 상당한 성능 향상 효과를 볼 수도 있다.  
  기본값 : false

### 리듀스 태스크 튜닝

파티셔너는 맵의 출력 파일을 리듀서 개수에 맞게 분할한다. 그리고 리듀스 태스크는 네트워크를 통해 자신에게 할당된 파일을 가져온 후, 병합정렬을 수행하고, 동일 키에 대해 그룹화를 한후, 리듀서 클래스로 전달한다. 마지막으로 리듀서는 최종 결과값을 출력하게 된다.

따라서 리듀스 태스크에서 성능향상을 할 수 있는 요소는 맵의 결과를 가져오는 부분과, 최종 결과를 저장하는 부분이다.

* mapred.reduce.tasks  
  [[ …]]
* mapred.reduce.max.attempts  
  리듀스 태스크 실해 후 재시도하는 횟수  
  기본값 : 4
* mapred.reduce.paralle.copies  
  맵의 출력 파일을 병롤로 가져오기 위한 동시 개수  
  기본값 : 5
* mapred.reduce.task.speculative.execution
* mapred.output.compress  
  리듀스의 출력 파일을 압축할지 여부여부  
  기본값 : false

### 병합 정렬 튜닝

[[ 필요한가?? ]]

### 잡트래커/태스크 트래커 튜닝

잡 트래커는 잡에 대한 실행을 관리하고, 각 태스크 트래커에 태스크를 할당한다. 각 태스크 트래커는 실제 맵 태스크나 리듀서 태스크를 실행하는 서버다. 따라서 잡 트래커의 튜닝 요소는 전체 시스템에 대한 설정이고, 태스크 트래커는 단일 서버에 대한 설정이다.

* mapred.max.tracker.failures  
  하나의 잡을 실행하면 어려개의 태스크가 실행된다. 이때 특정 태스크에 문제가 발생했을 때, 다시 시도하는 횟수  
  기본값 : 4
* mapred.job.tracker.handler.count  
  태스크 트래커의 요청을 처리하기 위한 잡 트래커의 소켓 서버 핸들러 개수  
  트래커 서버 수의 4%정도가 적당  
  기본값 : 10
* mapred.jobtracker.restart.recover  
  기본값 : false
* mapred.jobtracker.maxtasks.per.job  
  단일 작업에서 허용되는 최대 태스크 수.  
  기본값 : -1 (무제한)
* mapred.local.dir  
  태스크 트래커에서 태스크 실행에 필요한 임시 로컬 디렉토리에 대한 경로.   
  하둡 파일 시스템의 데이터가 저장되는 디렉토리와 다른 디렉토리를 지정하는 것이 좋다  
  기본값 : ${hadoop.impl.dir}/mapred/local
* local.cache.size  
  기본값 : 1GB
* mapred.local.dir.minspacestart  
  mapred.local.dir에 사용 가능한 디렉토리 공간이 설정된 값보다 작으면 더 이상 태스크를 실행하지 않는다. 이는 태스크 실행 중에 디스크 공간이 부족해서 실패하는 것을 미리 방지하기 위해서다  
  기본값 : 0
* mapred.map.task.maximum  
  하나의 태스크 트래커에서 동시에 실행할 수 있는 맵 태스크의 개수.  
  CPU 개수만큼 설정하거나 적게 설정해야 함  
  기본값 : 2
* mapred.reduce.task.maximum  
  하나의 리듀서 태스크에서 동시에 실행할 수 있는 리듀스 태스크의 개수  
  기본값 : 2
* tasktracker.http.threads  
  리듀서에서 맵 결과를 가져오기 위해 태스크 트래커로 HTTP 요청을 보내는데, 이때 이 요청을 처리하기 위한 HTTP 서버의 쓰레드 개수  
  기본값 : 40

## 맵리듀스 프로그래밍 활용 팁

== MapReduce Job 성공 여부 확인하기

MapReduce Job을 실행할 때 아래와 같은 2가지 명령어를 사용할 수 있다.

* **JobClient.runJob(job);**  
  Job을 실행 요청한 후, Job이 완료되어야만 그 다음 명령이 실행된다.
* **JobClient.submitJob(job);**Job을 실행 요청한 후, Job 완료 여부와 관계 없이 그 다음 명령이 실행된다.

따라서 Job의 성공 여부에 따라 실행할 다음 명령이 있는 경우 아래와 같이 pseudo 코드를 작성해 볼 수 있다.

* **JobClient.runJob(job); 의 경우**  
  *RunningJob runJob = jobClient.runJob(job);  
    
  // Job이 성공한 경우에만  
  if (****runJob.isSuccessful()****) {  
      // 원하는 다음 명령어를 실행한다.  
  }*
* **JobClient.submitJob(job); 의 경우**Job을 실행 요청한 후, Job 완료 여부와 관계 없이 그 다음 명령이 실행된다.  
  *RunningJob runJob = jobClient.submitJob(job);  
    
  // Job이 완료될 때까지 ㄱ  
  while (****!runJob.isComplete()****)  
      Thread.sleep(timeInMilliseconds);  
  }  
  // Job이 성공한 경우에만  
  if (****runJob.isSuccessful()****) {  
          // 원하는 다음 명령어를 실행한다.  
      }*

**[참고자료]**

* [JavaDoc : Interface RunningJob](http://hadoop.apache.org/docs/r0.20.0/api/org/apache/hadoop/mapred/RunningJob.html)
* [JavaDoc : class JobClient](http://hadoop.apache.org/docs/r0.20.0/api/org/apache/hadoop/mapred/JobClient.html)

== 하둡 프로그래밍 팁

**== FileSystem API ==**

**파일 삭제**

FileSystem.delete() 메서드에는 2가지 종류가 있다.

* delete(Path f)
* delete(Path f, boolean recursive)

만약 f가 디렉토리이고 디렉토리가 비어있지 않다면 f는 지워지지 않는다. 현재 delete(Path f) 메서드는 deprecated되었으며 delete(Path f, boolean recursive)를 사용해야 한다.

만약 f를 모두 지우고자 한다면

delete(f, true);

와 같이 사용해야 한다.

**디렉토리 사용량(du)**

특정 디렉토리에 포함된 파일의 용량(하위 디렉토리까지 포함하여)을 계산하려면 아래와 같이 listStatus() 메서드를 사용하여 FileStatus의 배열을 가져온 후 getLen() 메서드를 호출하여 파일의 사이즈를 얻은 후, 합계를 구한느 방식으로 구할 수 있다.

FileStatus[] dtPaths = fs.**listStatus**(new Path("디렉토리 경로"));

double sum = 0.0;

for (FileStatus dtPath : dtPaths) {

    sum = sum + dtPath .**getLen**();

}

하지만 이렇게 재귀적으로 용량을 계산하는 방법은 자원 소모가 크고 느리다. 이보다는 콘솔의 du 명령에 해당하는 getContentSummary() 메서드를 호출하여 ContentSummary 객체를 얻은 후, getLength() 메서드를 호출하는 편이 낫다. 주의할 점은 반환되는 값이 gb 단위의 long 타입이 반환되다는 점이다.

fs.**getContentSummary**(new Path("디렉토리 경로").**getLength**();

## 하이브 성능향상하기

== 하이브 쿼리 성능 튜닝

**병렬 처리**

하이브는 쿼리문을 하나 이상의 스테이지로 변환한다. 각 스테이지 사이에는 의존관계가 있는 경우도 있다. 기본적으로 하이브는 스테이지를 한번에 하나씩 처리한다. 하지만 의존관계가 없는 스테이지가 있다면, 동시에 병렬로 수행하게 되면 더 빨리 처리할 수 있다.

병렬처리를 사용하려면 **hive.exec.parallel**을 true로 설정한다.

hive> set hive.exec.parallel = true;

단 병렬처리를 사용하면 하둡 클러스터 자원을 더 많이 사용하게 된다.

**LIMIT 설정**

최적의 쿼리문을 만드는 중간 과정에서 빠른 결과를 보기 위하여 LIMIT 절을 사용하는 경우가 많다. 하지만 LIMIT절을 사용하더라도 데이터 전체에 대해 쿼리문을 수행 한후, LIMIT절에 기술된 수만큼 결과를 가져올 뿐이다. 따라서 성능 향상 효과는 미미하며, 오직 콘솔에 출력되는 시간만을 단축할 수 있을 뿐이다.

만약 LIMIT절을 이처럼 중간과정에서 사용하는 용도라면, 쿼리문이 전체 데이터가 아닌 일부 데이터만 적용하도록 설정할 수 있다.

|  |  |  |
| --- | --- | --- |
| 설정 키 | 설정값 | 설명 |
| hive.limit.optimize.enable | true | LIMIT절 사용시, 일부 데이터만 탐색하도록 설정할 것인지 여부 |
| hive.limit.row.max.size | 100000 | LIMIT 최적화 사용시, 최대로 볼 수 있는 행의 개수 |
| hive.limit.optimize.limit.file | 20 | LIMIT 최적화 사용시, 최대로 볼 파일의 개수 |

**STRICT 모드 설정**

사용자가 잘못된 쿼리문을 실행하여 잡트래커의 자원의 대부분을 낭비하게 되는 경우를 방지하기 위해, 하이브 실행 모드를 strict로 설정할 수 있다. 기본값은 nonstrict다.

hive> set hive.mapred.mode = strict;

STRICT 모드로 설정하면 아래의 쿼리문을 실행할 수 없다.

* 파티션이 있는 테이블인 경우, 파티션 필터를 사용하지 않는 쿼리문
* ORDER BY절이 있는 경우, LIMIT절이 없는 쿼리문
* 카테시안 프로덕트를 생성하는 쿼리문

**리듀서 수의 최적화**

리듀스가 필요한 쿼리문을 실행한 경우 리듀스 개수를 설정하지 않았다면, 하이브는 입력 데이터의 크기를 기반으로 리듀스의 개수를 결정한다. 예를 들어 입력 데이터가 2GB인 경우, 리듀서의 개수는 2개가 되는데, 1GB당 1개의 리듀서가 생성되기 때문이다. 이러한 설정 값은 **hive.exec.reducers.bytes.per.reducer**로 설정할 수 있다.

대개의 경우는 기본값(1GB)으로도 충분하지만, 맵단계의 출력값, 즉 리듀서의 입력값이 맵의 입력값에 비해 훨씬 많은 경우라면 더 많은 리듀서가 필요하다. 또는 반대로 리듀서의 입력값이 맵의 입력값에 비해 훨씬 적은 경우라면, 더 적은 리듀서가 필요하다. 이러한 경우라면 **mapred.reduce.tasks** 옵션을 통해 직접 리듀서의 개술르 설정할 수 있다.

하지만 여러명의 사용자가 함께 사용중인 클러스터러면, 하나의 쿼리가 너무 많은 자원을 소모하지 않도록 방지할 필요가 있다. **hive.exec.reducers.max**를 임시가 아니라 hive-site.xml에 설정하여, 동시에 사용하는 사용자의 사용성을 보장해야 한다(물론 FairScheduler와 같이 맵리듀스 잡이 동시에 실행되도록 설정한 경우에만).

**JVM 재사용**

입력 파일이 작고 실행하려는 태스크가 단순하거나 또는 매퍼와 리듀서를 초기화하는데 시간이 오래 걸린다면, 태스크 자체의 실행시간보다 각 태스크를 위한 JVM을 생성하고 초기화하는데 더 많인 시간이 소모될 수도 있다. 하둡 0.19 버전부터 동일한 잡의 여러 태스크에서 하나의 JVM을 재사용할 수 있는 옵션이 추가되었다. **mapred.job.reuse.jvm.num.tasks** 속성에 재사용할 횟수를 설정하면 된다. 기본값을 1로, 전혀 재사용하지 않는다.

문제는 자바 가상 머신을 재사용하도록 설정하면, 해당 잡을 위해 예약된 태스크 슬롯은 잡이 종료되어야만 해제한다는 점이다. 만약 해당 잡의 특정 태스크가 다른 태스크보다 상당히 오래 걸려서 결국 잡 전체의 실행시간이 지연되는 경우라면, 해당 잡에 할당된 태스크 슬롯을 나머지 잡에서 사용하지 못하게 된다.

== Group By 절의 성능 향상을 위한 맵 사이드 집계

하이브에서 count, avg와 같은 집계함수를 사용할 때, 성능 향상을 위한 옵션으로 **hive.map.aggr** 속성이 있다. 아래와 같이 이 속성을 true로 설정하면, 집계함수 사용시 맵 사이드에서 집계가 추가적으로 이루어진다.

hive> set hive.map.aggr = true;

기본적인 원리는 맵 단계의 임시 결과를 리듀서로 보내기 전에 컴바이너를 호출하는 방식과 동일하다. 즉 컴바이너를 호출함으로써 리듀서로 나가는 출력을 줄임으로써, I/O 측면에서 성능향상 효과를 볼 수 있다. 하지만 메모리 사용량이 추가적으로 늘어나게 된다.

아래는 하이브 위키에서 설명하는 내용이다.

**"Group By절을 위한 맵 사이드 집계"**

hive.map.aggr 옵션은 집계를 하는 방식을 제어한다. 기본값은 false다. true로 설정하면, 하이브는 맵 태스크에서 직접 일단계 집계를 수행하게 된다.

***\*\* 현재 사용중인 0.8.1 버전에서는 true가 기본 값이다.***

이를 통해 약간의 성능을 향상시킬 수 있지만, 성공적으로 실행하려면 더 많은 메모리를 필요로 한다.

아래는 대런 리(Darren Lee)가 bizo 블로그에서 설명하는 내용이다.

**"아파치 하이브: 맵 사이드 집계"**

하이브에서 대용량의 데이터에 대해 집계 함수를 실행할 때, 흔히 마주치게 되는 예러는 다음과 같다.

에러:

  맵사이드 집계에서 사용하는 해시 맵으로 인한 Out of memory

해결책:

     hive.map.aggr.hash.percentmemory는 기본적으로 0.5로 설정된다. 0.5보다 작은 값으로 설정한다.

     예를 들어 'set hive.map.aggr.hash.percentmemory = 0.25;'와 같이 설정한다.

이 문제는 하이브가 맵사이드 집계를 실행하여 쿼리를 최적화하려고 할 때 발생한다. 맵사이드 집계란 매퍼 내부에서 집계를 부분적으로 실행하여 맵 단계에서 최적화를 하는 과정으로, 이를 통해 매퍼는 더 적은 수의 행을 생성하게 된다. 결과적으로 하둡에서 정렬하거나 리듀서에 분배해야 하는 정보의 량이 줄어드는 효과를 만들어 낸다.

흔히 사용하는 "단어 세기(word count)" 예제를 통해 하둡 잡이 어떻게 실행되는지 살펴보자.

단어 세기 예제의 경우, 기본적으로 매퍼는 입력에서 각 행을 토큰으로 분리한 후, **(#{token}, 1)**와 같이 키-값 쌍의 형태로 출력을 만들어낸다. 하둡 프레임워크는 이러한 키-값 쌍을 토큰별로 정렬한 후, 리듀서에서는 각 토큰에 대한 전체 갯수를 출력하기 위해 값 값에 대한 합계를 계산한다.

반면 맵사이드 집계를 사용하게 되면, 매퍼에서는 각 행을 토큰 단위로 분리할 뿐만 아니라 메모리에 해시 맵을 생성하여 각 토큰에 대한 부분 합계를 저장해 둔다. (더 정확히 말하자면, 매퍼는 키와 대응하는 부분 집계값, 즉 키의 개수를 저장한다.) 그런 후 매퍼는 **(#{token}, #{token\_count})** 쌍을 출력으로 만들어낸다. 이경우에도 마찬가지로 하둡 프레임워크에서는 매퍼의 출력값을 정렬하며, 리듀서에서는 각 토큰별로 합계를 계산하여 전쳬 합계를 출력으로 만들어낸다. 따라서 매퍼는 각 토큰이 나타날 때마다 매번 한 행을 출력하는 게 아니라, 토큰 하나당 오직 한 행만이 출력으로 만들어진다. 이로 인해 모든 토큰에 대한 맵을 메모리에 유지해야 한다는 단점이 있다.

기본적으로 하이브는 맵사이드 집계를 사용하려고 시도한다. 맵사이드 집계 중에 사용중인 해시 맵이 메모리를 너무 많이 차지하게 되면, 맵사이드 집계를 중단하고 기본적인 방식으로 실행되게 된다. 좀더 자세히 말하면, 1000,000행(**hive.groupby.mapaggr.checkinterval**를 통해 설정 가능)을 처리한 후 하이브는 해시맵에 있는 항목 수를 검사한다. 만약 항목 수가 현재까지 읽은 행수의 50%(**hive.map.aggr.hash.min.reduction**로 설정 가능)보다 많으면, 맵사이드 집계가 중단된다.

뿐만 아니라 하이브는 해시맵의 각 항목마다 요구하는 메모리 량을 측정한 후, 맵의 사이즈가 매퍼에서 사용가능한 메모리 총량의 50%(**hive.map.aggr.hash.percentmemory**로 설정 가능)보다 많아지면, 맵의 결과를 리듀셔로 바로 보낸다. 하지만 이러한 과정은 행의 개수와 각 행의 사이즈를 하이브에서 추정한 결과에 따른 것이므로, 각 행별로 실제로 차지하는 메모리 사용량이 추정치보다 높아지면, 해시 맵의 데이터가 리듀서로 보내지기도 전에 매퍼에서 Out of Memory 에러가 발생하게 된다.

특히 count distinct 집계를 호출하는 쿼리를 사용하는 경우에는, 부분 집계는 실제로 모든 값에 대한 목록을 포함하게 된다. 유니크한(즉 distinct한) 키에 대해 값이 많이 있다면, 해시맵의 행의 개수는 늘지 않더라도 맵에서 사용하는 메모리 총량은 증가하게 된다. 하지만 앞서 말했듯이 부분적인 집계 결과를 리듀서로 출력할지 결정하는 기준은 해시맵의 행의 개수다.

맵사이드 집계가 true로 설정된 상태에서 group by절을 포함하는 쿼리문을 실행할 때, 매퍼에서 메모리가 부족해지는 현상이 발생하면, 하이브에서는 메모리 부족 현상을 피하기 위해 리듀서로 출력하는 기준을 더 낮출 것을 제안하는 도움말을 표시해준다. 따라서 하이브가 맵의 출력을 리듀서로 보내는 기준(행의 개수)를 낮출 수 있다. 하지만 이렇게 하더라도 행의 개수와는 무관하게 맵의 사이즈(바이트 단위)가 증가하게 되는 문제를 해결할 수는 없다.

대안으로는 맵사이드 집계를 아예 사용하지 않을 수도 있다(set hive.map.aggr = false). 또는 하둡 설정을 통해 매퍼에서 사용할 메모리르 좀더 증가시키거나, 하이브가 쿼리 실행 계획을 다르게 세우도록 쿼리문을 달리 작성할 수도 있다.

예를 들어 아래과 같은 단둔한 쿼리문은

select count(distinct v) from tbl

아래와 같이 작성할 수 있다.

select count(1) from (select v from tbl group by v) t

이렇게 하면 count distinct 집계를사용하지 않으므로, 더 효율적으로 실행될 수도 있다.

**[참고자료]**

* [하이브 위키, Map-side Aggregation for Group By](https://cwiki.apache.org/Hive/languagemanual-groupby.html)
* [Map-side aggregations in Apache Hive](http://dev.bizo.com/2013/02/map-side-aggregations-in-apache-hive.html), Darren Lee

== 하이브 결과 압축하기

**개요**

하이브 퀴리를 실행하면 결국 맵리듀스 잡으로 변환되어 잡트래커에 제출되므로, 하이브 쿼리의 결과를 압축할 수 있는 부분은 총 2군데이다. 맵의 중간결과를 압축하는 부분과 리듀스의 출력, 즉 잡의 최종 결과를 압축하는 부분이다.

맵리듀스 잡의 결과를 압축하게 되면 오히려 압축과정으로 인해 처리속도가 떨어지지 않을까 하는 우려도 있다. 하지만 하둡은 CPU보다는 I/O를 더 많이 쓰는 경향이 있다. 따라서 실행하려는 하이브 쿼리문이 CPU 연산보다는 입/출력에 더 많은 자원을 소모한다면 출력을 압축하는 편이 더 빨리 실행될 수 있겠다.

**압축 코덱 선택**

하둡 최신 버전은 기본적으로 GzipCode, Bzip2Codec과 DefaultCodec을 지원한다. 현재 사용중인 하둡에서 지원하는 코덱을 확인하려면 아래의 명령을 실행한다. 별다른 설정을 하지 않았다면 ${HADOOP\_HOME}/src/core/core-default.xml 파일에서(또는 core-site.xml과 같은 파일에서) **io.compression.codecs**설정을 확인한다.

<name>io.compression.codecs</name>

<value>org.apache.hadoop.io.compress.**DefaultCodec**,

            org.apache.hadoop.io.compress.**GzipCodec**,

            org.apache.hadoop.io.compress.**BZip2Codec**</value>

<description>A list of the compression codec classes that can be used for compression/decompression.</description>

</property>

물론 필요하다면 원하는 압축 코덱을 얼마든지 선택할 수 있다(단 하둡을 재기동해야 한다. 실시간으로 데이터를 수집중인 환경에서 네임노드가 이중화되지 않은 상황이라면 선택권이 없다).

이외에도최근에는 Snappy 압축 코덱과 LZO 압축 코델을 많이 사용한다. 각 압축 코덱의 장단점은 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DefaultCodec | GzipCodec | Bzip2Codec | Snappy | LZO |
| 압축률 |  | 높음 | 매우 높음 | 낮음 | 낮음 |
| 압축속도 |  | 중간 | 낮음 | 빠름 | 빠름 |
| 분할압축 |  | X | O | X | O |

압축 코덱은 실행하려는 맵리듀스 잡의 성격에 따라 달리 선택해야 한다. 예를 들어 맵리듀스 잡의 결과를 다른 맵리듀스 잡의 결과로 활용하려면 반드시 분할 압축을 지원하는 코덱을 선택해야 한다.

반면 하이브 쿼리의 실행 결과는 영구적이기 보다는 임시용인 경우가 많으므로(CTAS문이 아닌 SELECT문인 경우에는 더더욱), 분할압축이 지원되지 않더라도 문제가 없다. 대신 압축률과 압축속도가 더 중요한 기준이 된다. 일반적으로 아래와 같이 선택해볼 수 있겠다.

|  |  |  |
| --- | --- | --- |
| 위치 | 압축 코덱 | 이유 |
| 맵의 임시 결과 압축 | Snappy | 중간 과정 압축은 맵에서 리듀서로 보내는 데이터 량을 줄일 수 있다. 리듀서에서 다시 압축을 해제해야 하므로, 압축률보다는 압축속도가 빠른 코덱이 더 나은 선택이다. |
| 잡의 최종 결과 압축 | GzipCodec | Gzip 압축은 압축률이 높으므로 출력 압축으로 좋다. 하지만 압축된 결과를 하이브 CLI에서 직접 보기 위한 용도가 아니라면, 최종 결과는 압축하지 않는 편이 낫겠다. |

**압축 코덱 사용하기**

하이브에서 압축 코덱을 사용하려면 아래와 같이 설정할 수 있다(사용중인 하둡 클러스터에 Snappy 코덱이 설치되어 있지 않은 관계로 임시 결과 압축에도 GzipCodec을 사용했다).

|  |  |
| --- | --- |
| 위치 | 설정 |
| 맵의 임시  결과 압축 | **mapred.map.output.compression.codec**=org.apache.hadoop.io.compress.GzipCodec  **hive.exec.compress.intermediate**=true |
| 잡의 최종   결과 압축 | **mapred.output.compression.codec**=org.apache.hadoop.io.compress.GzipCodec  **hive.exec.compress.output**=true |

실행하려는 하이브 쿼리문에 따라 압축할지 선택해야 하므로, CLI에서 매번 설정해야 한다. 이보다는 .hiverc. 파일을 사용하여 hive CLI에 대한 알리아스를 지정하여 사용하면 더욱 간편할 듯 하다.

우선 하이브 사용자의 홈 디렉토리에는 기본설정이 된 .hiverc를 만든다.

set hive.cli.print.current.db=true;

set hive.cli.print.header=true;

그리고 임시 결과 압축을 사용할 때 사용할 설정 파일과, 최종 결과까지 압축할 때 사용할 설정 파일을 .hive/.hiverc\_map\_zip과 .hive/.hiverc\_out\_zip 파일로 각각 만든다.

[.hive/.hiverc\_map\_zip]

set hive.cli.print.current.db=true;

set hive.cli.print.header=true;

set mapred.map.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;

set hive.exec.compress.intermediate=true;

[.hive/.hiverc\_out\_zip]

set hive.cli.print.current.db=true;

set hive.cli.print.header=true;

set mapred.map.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;

set hive.exec.compress.intermediate=true;

set mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;

set hive.exec.compress.output=true;

이제 .bashrc 파일에 각각의 파일에 대한 알리아스를 생성한다.

alias mive='/home/hive/hive/bin/hive -i /home/hive/.hive/.hiverc\_map\_zip'

alias nive='/home/hive/hive/bin/hive -i /home/hive/.hive/.hiverc\_out\_zip'

== 하이브 CLI 활용 팁

**쿼리문 편집시 이동 단축키**

* Ctrl + A : 맨 앞으로
* Ctrl + E : 맨 뒤로
* TAB : 예약어 및 함수 자동완성

쿼리문 편집시 TAB키가 자동완성으로 사용되기 때문에, 쿼리문에 TAB키가 포함된 경우 쿼리문이 정상적으로 실행되지 않는다. 대개의 경우 TAB키를 쿼리 포맷팅용으로 많이 활용하기 때문에 간혹 번거로울 때가 있다. 하이브에서 자동완성 키를 수정할 수 있는 방법이 있는지는 아직 찾지 못했는데 혹시 아시는분은 댓글로 부탁드립니다.

**하이브 CLI에서 쉘 명령어 실행하기**

명령문 앞에 ! 를 입력한 후, 세미콜론으로 마무리한다.

hive>! ls /

주의할 점은 쉘 파이프와 파일 글로빙은 지원하지 않는다.

**하이브에서 하둡 dfs 명령어 실행하기**

명령문 앞에 dfs 를 입력한 후, 세미콜론으로 마무리한다.

hive>dfs -ls /

쉘에서 하둡 dfs 명령문을 실행하는 것에 비해 하이브 CLI에서 하둡 dfs 명령문을 실행하면 별도의 JVM 인스턴스를 구동하지 않으므로 좀 더 빨리 실행된다.

== 하이브 활용 팁 - CLI 프롬프트와 .hiverc 파일

하이브를 사용하다 보면 기본 프롬프토가 아니라, 현재 사용중인 데이터베이스 명이 나오면 더 편하지 않을까 생각해 본적 있다면 이 글이 도움이 될 것이다.

아무런 설정도 하진 않는 경우 하이브 CLI를 실행하면 아래와 같인 기본 프롬프트가 나타난다.

hive>

만약 기본 프롬프트가 아니라, 현재 사용중인 데이터베이스가 프롬프토에 보이게 하고 싶다면 아래와 같이 설정한다.

hive>**set hive.cli.print.current.db=true;**

hive (test)>

또한 하이브 쿼리를 실행한 경우, 헤더가 표시되지 않아 제대로 실행이 된것인지 의문이 될 때가 많다. 하이브 쿼리문의 실행 결과에 헤더를 표시하고 싶다면 아래와 같이 설정한다.

hive (test)> set hive.cli.print.header=true;

물론 이렇게 설정하더라도 필드 단위의 정렬은 되지 않으므로 보기에는 약간 번거롭기는 하다.

하지만 이렇게 설정하면 매번 접속할 때마다 설정해야 하는 번거로움이 있다. 이때 .hiverc 파일을 활용한다.

하이브는 CLI를 실행할 때마다 자동으로 $HIVE\_HOME/.hiverc 파일에 설정된 명령을 실행하게 된다. 따라서 .hiverc 파일에 기본적으로 필요한 옵션을 필요한 명령을 설정하면 된다.

== 하이브 로컬 모드

하둡이 분산 모드로 운영되더라도, 하이브 쿼리를 실행할 때 로컬 모드로 실행할 수 있다. 하이블 쿼리를 로컬 모드로 실행하게 되면, 하이브가 직접 데이터 파일을 읽고 맵리듀스 태스크를 관리하기 때문에, 더 빨리 실행된다. 단 입력 데이터가 작은 경우에만 로컬 모드로 실행해야 더 빠른 실행 속도를 얻을 수 있다.

아래는 하이브 위키에서 설명하는 내용이다.

일반적으로 하이브 쿼리문은 하이브 컴파일러에 의해 맵리듀스 잡으로 생성된다. 이렇게 생성된 잡은 **mapred.job.tracker** 변수에 설정된 맵리듀스 클러스터로 보내진다.

대개의 경우 mapred.job.tracker 변수는 다수의 노드로 구성된 맵리듀스 클러스터에 해당한다. 하지만 하둡에서는 맵리듀스 잡을 로컬 장비에서 간단히 실행할 수 있는 옵션을 제공한다. 작은 데이터셋에 대해 쿼리문을 실행할 경우, 이 옵션을 사용하면 꽤 효과적이다. 실제로 데이터셋이 작은 경우 맵리듀스 잡을 하둡 클러스터로 보내는 경우에 비해, 로컬 모드로 실행하는 편이 훨씬 더 빠른다. 이때 데이터는 .HDFS로부터 투명하게 접근된다. 반대로 말하면, 로컬 모드에서는 단 하나의 리듀서만 실행하므로, 데이터셋이 큰 경우 훨씬 더 느려질 수 있다.

하이브 0.7 버전부터는 로컬 모드 실행을 완벽히 지원한다. 로컬 모드를 사용하려면 아래의 옵션을 설정해야 한다.

hive> SET mapred.job.tracker=local;

또한 **mapred.local.dir**이 로컬 장비에 실제로 위치한 경로를 가리키도록 설정해야 한다(예를 들어 /tmp/<username>/mapred/local와 같이). 만약 설정하지 않는다면 로컬 디시크 공간을 찾을 수 없다는 예외를 나타날 것이다.

뿐만 아니라 하이브 0.7 버전부터는 맵리듀스 잡을 자동으로 로컬 모드로 실행할 수 있는 옵션을 제공한다. 자동으로 실행하려면 아래와 같이 설정한다.

hive> SET hive.exec.mode.local.auto=true;

일반적으로 자동 로컬 모드는 비활성화되어 있다. 자동 로컬 모드를 활성화하면, 하이브는 각 쿼리문으부터 생성된 맵리듀스 잡의 크기를 분석한 후, 아래의 조건을 만족하면 자동으로 로컬모드로 실행하게 된다.

* 잡의 전체 입력 사이즈 < hive.exec.mode.local.auto.inputbytes.max (디폴트, 128MB)
* 맵 태스트 전체 개수    < hive.exec.mode.local.auto.tasks.max (디폴트 4)
* 리듀스 태스크 전체 개수 < 2

따라서 데이터셋이 작거나, 다수의 맵리듀스 잡이 실행되는 경우에도 연속된 잡의 입력이 상당히 작다면, 잡은 로컬 모드로 실행된다.

주의할 점은 하둡 서버 노드의 실행 환경과 하이브 클라이언트가 실행되는 장비의 환경은 다를 수 있다는 점이다(예를 들어 jvm 버전이 다르다거나, 다른 버전의 소프트웨어 라이브러리를 쓴다거나). 이로 인해 로컬 모드로 실행한느 경우 생각치 못한 방식으로 동ㅈ가하거나 에러가 발생할 수도 있다. 또한 주의할 사항은 로컬 모드 실행은 별도의 (하이브 클라이언트의) 자식 jvm으로 실행된다는 점이다. 원한다면 자식 jvm이 사용가능한 최대 메모리 크기를 **hive.mapred.local.mem** 옵션을 통해 설정할 수 있다. 기본적으로 메모리 크기는 0으로 설정되어 있다. 기본값으로 설정된 경우, 자식 jvm이 사용간으한 메모리 한계를 하둡이 결정하도록 둔다.

**[참고자료]**

* [하이브 위키, Getting STarted, Hive, Map-Reduce and Local-Mode](https://cwiki.apache.org/Hive/gettingstarted.html)
* 하이브 완벽 가이드, 한빛 미디어, 오세봉 외 옮김

## 검색 인덱스 관라하기

# 부록

## 부록 1 웹로그 포맷

원문 : http://publib.boulder.ibm.com/tividd/td/ITWSA/ITWSA\_info45/en\_US/HTML/guide/c-logs.html

**로그 파일 포맷**

[Web Site Analyzer](http://publib.boulder.ibm.com/infocenter/wchelp/v5r6/index.jsp?topic=/com.ibm.commerce.wca.doc/concepts/caztwsa.htm)는 HTTP 서버, FTP 서버 등의 서버 로그 파일에 포함된 정보를 활용하여 사이트를 분석한다. Web Site Analyzer에서 분석할 수 있는 로그 포맷은 다음과 같다.

* NCSA 로그(Common or Access, Combined, and Separate or 3-Log)
* W3C 확장 로그(Microsoft IIS 4.0과 5.0에서 사용)
* SunTM ONE 웹서버 로그 (iPlanet)
* IBM Tivoli Access Manager WebSEAL
* WebSphere 어플리케이션 서버 로그
* FTP 로그
* 커스텀 로그 파일 포맷(사용자가 필드를 직접 정의)

만약 여러분의 사이트에서 사용하는 로그 포맷이 표준을 따르지 않거나, FTP 로그를 분석해야 하는 경우라면 커스텀 로그 포맷을 직접 만들어야 한다.

또한 커스텀 로그 포맷을 직접 정의했다면, 커스텀 로그를 분석하는 파서는 직접 만들어야 한다(상세한 정보는 [커스텀 함수 만들기](http://publib.boulder.ibm.com/tividd/td/ITWSA/ITWSA_info45/en_US/HTML/guide/t-extend.html)부분을 참고하라).

주의할 점

Web Site Analyzer는 2바이트 또는 아스키 코드가 아닌 문자로 구성된 로그 파일을 변환하려고 시도하지 않는다. 만약 2바이트로 구성된 문자열을 포함하는 레코드를 임포트한 후 이 데이터를 바탕으로 보고서를 생성하려고 하면, 2바이트 문자열은 제대로 표시되지 않게 된다. 이 문제를 해결하려면 여러분의 웹서버가 UTF-8을 사용하여 로그 파일을 생성하도록 설정해야 한다.

Web Site Analyzer는 로그 파일의 cookie와 user-agents 필드에 있는 + 기호를 공백 문자로 처리하는 방식으로 파싱하게 된다. 또한 각 키와 값에 포함된 맨 앞 또는 맨 뒤의 공백을 제거한다.

**지원하는 서버의 종류**

Web Site Analyzer는 아래의 서버에서 생성하는 로그를 파싱할 수 있다.

* HTTP Server
* FTP Server
* Supplemental Server
* IBM WebSphere Portal
* IBM WebSphere Personalization
* IBM WebSphere Edge Server
* SunTM ONE Web Server (iPlanet) Server

**NCSA 로그 포맷**

NCSA 로그 포맷은 NCSA httpd을 기반으로 하며, 대다수의 HTTP 서버 벤더에서 표준으로 자리잡고 있다.

Web Site Analyzer는 아래의 형태를 지원한다.

* Common (access log)
* Combined
* Separate with Date (three log)

HTTP 서버에 따라 사용할 수 있는 로그 포맷의 형태는 달라진다.

**NCSA Common (access log)**

NCSA Common 로그 포맷은 가장 기본적인 HTTP 접근 정보만을 포함한다. NCSA Common 로그는 흔히 Access 로그라고도 부르며, NCSA에서 분리하는 로그 포맷의 세가지 종류 중 하나다. 또한 Common 로그 포맷은 referral와 user-agent 정보가 없는 NCSA Combined 로그 포맷으로도 볼 수 있다.

Common 로그는 요청받은 자원을 포함하여 몇가지 정보를 포함한다. 하지만 referral, user-agent, cookie 정보는 포함하지 않는다. 이들 정보는 모두 하나의 파일에 저장된다.

Common 로그 파일은 아래와 같은 형태로 필드가 저장된다.

host rfc931 username date:time request statuscode bytes

아래는 Common 로그 파일에 기록되는 정보의 예시다.

125.125.125.125 - dsmith [10/Oct/1999:21:15:05 +0500] "GET /index.html HTTP/1.0" 200 1043

위의 Common 로그 포맷에 대해, 각 필드에 대한 자세한 설명은 아래와 같다.

* **host**(예시의 경우, 125.125.125.125)  
  HTTP 요청을 한 HTTP 클라이언트의 IP 주소 또는 호스트/서브도메인의 이름
* **rfc931** (예시의 경우, "-")  
  HTTP 요청을 한 클라이언트를 식별하기 위해 사용되는 식별자. 만약 정보가 없다면 "-"로 기록된다.
* **username** (예시의 경우, dsmith)  
  클라이언트에서 인증을 위해 사용한 사용자명 또는 사용자 id. 만약 정보가 없다면 "-"로 기록된다.
* **date:time timezone** (예시의 경우, [10/Oct/1999:21:15:05 +0500])  
  HTTP 요청을 한 일시 또는 타임스탬프.  
  data/time 필드는 아래와 같은 형태로 기록된다.  
  **[dd/MMM/yyyy:hh:mm:ss +-hhmm]**  
  각 필드의 의미는 다음과 같다.
  + dd  
    일
  + MMM  
    월
  + yyyy  
    년도
  + :hh  
    시간
  + :mm  
    분
  + :ss  
    초
  + +-hhmm  
    타임존(시간대)

일반적으로 일(day)은 한자리의 숫자더라도, 2자리의 숫자 형태로 기록된다. 예를 들어 해당 월의 둘째날이더라도 02와 같이 기록된다. 하지만 일부 HTTP 서버에서는 1자리 숫자의 날짜는 1자리로 기록하기도 한다. 따라서 로그를 파싱할 때, 어떤 형태의 로그든 파싱할 수 있어야 한다.

* **request** (예시의 경우, "GET /index.html HTTP/1.0")  
  HTTP 요청. 이 필드에는 세가지 종류의 정보가 포함된다. 가장 중요한 정보는 요청받은 자원(index.html)이다. 이외에도 HTTP 메서드(GET)와 HTTP 프로토콜(1.0) 정보가 함께 기록된다.
* **statuscode** (예시의 경우, 200)  
  상태 코드는 HTTP 요청이 성공했는지, 아니면 실패했는지를 나타내는 숫자다.
* **bytes** (예시의 경우, 1043)  
  바이트 필드는 HTTP 요청에 대한 응답으로 전성된 데이터의 바이트 사이즈를 포함한다. 이떄 HTTP 헤더의 크기는 포함하지 않는다.

**NCSA Combined**

NCSA Combined 로그 포맷은 NCSA Common 로그 포맷을 확장한 형식이다. Combined 로그 포맷은 Common 로그 포맷과 동일한 정보를 포함할 뿐만 아니라, 선택적으로 referral, user-agent, cookie 필드와 같은 세가지 정보를 추가적으로 가진다.

Combined 로그 파일은 아래와 같은 형태로 필드가 저장된다(추가된 필드는 굵게 표시했다).

host rfc931 username date:time request statuscode bytes **referrer user\_agent cookie**

예시

125.125.125.125 - dsmith [10/Oct/1999:21:15:05 +0500] "GET /index.html HTTP/1.0" 200 1043 **"http://www.ibm.com/" "Mozilla/4.05 [en] (WinNT; I)" "USERID=CustomerA;IMPID=01234"**

추가된 필드에 대한 설명은 아래와 같다.

* **referrer**(예시의 경우, "http://www.ibm.com/")  
  사용자가 여러분의 사이트에 접속시 사용한 링크의 URL(선택 필드)
* **user\_agent** (예시의 경우, "Mozilla/4.05 [en] (WinNT; I)")  
  방문자가 여러분의 사이트를 접속할 때 사용한 웹 브라우저와 플랫폼(선택 필드)
* **cookies**(예시의 경우, "USERID=CustomerA;IMPID=01234")  
  HTTP 서버가 클라이언트로부터 요청을 받았을 때, 클라이언트에게 다시 보내주는 일부 정보. 클라이언트 브라우저는 쿠키 정보를 임시로 저장하였다가 사용자가 HTTP 서버에 다시 요청을 할 때, 해당 요청에 추가되어 전송되게 된다. HTTP 서버는 각 HTTP 요청마다 여러개의 쿠키를 사용할 수 있다.  
  쿠키는 키=값 형태를 가진다. 키=값으로 구성된 다수의 쿠키는 세미콜론(;)으로 구분된다.   
  HTTP 서버에서 로그에 쿠키를 남기도록 설정하면, HTTP 서버가 요청을 받을떄마다 해당 정보를 로그에 기록하게 된다.

**Combined 형식으로 로그를 남기도록 웹서버 설정하기**

만약 IBM HTTP 서버나 Apache 버전의 웹서버를 사용중이라면, HTTP 설정 파일에서 해당 부분에 대한 지시문에서 주석을 없애는 방식으로 Combined 로그 포맷을 활성화할 수 있다.LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined

# If you prefer a single log file with access, agent and referrer information

# (Combined Logfile Format) you can use the following directive.

CustomLog logs/combined.log combined

**NCSA Seperate(3종류의 로그 포맷)**

NCSA Seperate 로그 포맷은 흔히 3종류의 로그 포맷으로도 부른다. 특정 로그 포맷에 따라 생성되는 로그 정보를 하나의 파일에 기록하는 대신에 별도의 파일(로그)에 기록할 수 있기 때문이다. 3종류의 로그 포맷은 아래와 같은 이름으로 부른다.

* Common log or access log
* Referral log
* Agent log

The three-log format contains the basic information in the NCSA Common log format in one file, and referral and user agent information in subsequent files. However, no cookie information is recorded in this log format.  
  
Common or access log  
The first of the three logs is Common log, sometimes referred to as the access log, which is identical in format and syntax to the NCSA Common log format.  
  
Referral log  
The referral log is the second of the three logs. The referral log contains a corresponding entry for each entry in the common log.  
  
The fields in the Referral log are:  
  
date:time referrer  
  
The following is an example of a record in a referral log:  
  
[10/Oct/1999:21:15:05 +0500] "http://www.ibm.com/index.html"  
The following is a description of the fields in the Referral log:  
  
date:time timezone ([10/Oct/1999:21:15:05 +0500] in the example)  
The date and time stamp of HTTP request. The date and time of an entry logged in the referral log corresponds to the resource access entry in the common log. As a result, the date and time of corresponding records from each of these logs will be the same. The syntax of the date stamp is identical to the date stamp in the common log.  
  
referrer ("http://www.ibm.com/index.html" in the example)  
The referrer is the URL of the HTTP resource that referred the user to the resource requested.  
  
For example, if a user is browsing a Web page such as http://www.ibm.com/index.html and the user clicks on a link to a secondary page, then the initial page has referred the user to the secondary page. The entry in the referral log for the secondary page will list the URL of the first page (http://www.ibm.com/index.html) as its referral.  
  
Agent log  
  
The Agent log is the third of the three logs making up the three-log format. Like the referral log, the agent log contains a corresponding entry for each entry in the common log.  
  
The fields in the Agent log are:  
  
date:time agent  
  
The following is an example of a record from an Agent log:  
  
[10/Oct/1999:21:15:05 +0500] "Microsoft Internet Explorer - 5.0"  
The following is a description of the fields in the Agent log:  
  
date:time timezone ([10/Oct/1999:21:15:05 +0500] in the example)  
The date and time stamp of HTTP request. The date and time of an entry logged in the agent log corresponds to the resource access entry in the common log. Because information logged in the agent log supplements information logged in the common log, the date and time of corresponding records from each of these logs will be the same. The syntax of the date stamp is identical to the date stamp in the Common log.  
  
  
agent "Microsoft Internet Explorer - 5.0" in the example)  
An HTTP client that makes HTTP requests. It is customary for an HTTP client, such as a Web browser, to identify itself by name when making an HTTP request. It is not required, but most HTTP clients do identify themselves by name. The Web server writes this name in the agent log.  
W3C Extended Log Format  
      This log file format is used by used by Microsoft Internet Information Server (IIS) 4.0 and 5.0.  
  
A log file in the extended format contains a sequence of lines containing ASCII characters. Each line may contain either a directive or an entry.  
  
Entries consist of a sequence of fields relating to a single HTTP transaction. Fields are separated by white space. If a field is unused in a particular entry dash "-" marks the omitted field.  
  
Directives record information about the logging process itself. Lines beginning with the # character contain directives. The following directives are defined in the W3C Extended format:  
  
Version: <integer>.<integer>  
  
The version of the extended log file format used. This draft defines version 1.0.  
  
Fields: [<specifier>...]  
  
lists a sequence of field identifiers specifying the information recorded in each entry.  
  
Software: string  
  
Identifies the software which generated the log.  
  
Start-Date: <date> <time>  
  
The date and time at which the log was started.  
  
End-Date:<date> <time>  
  
The date and time at which the log was finished.  
  
Date:<date> <time>  
  
The date and time at which the entry was added.  
  
Remark: <text>  
  
Comment information. Data recorded in this field should be ignored by analysis tools.  
  
The directives Version and Fields are required and should precede all entries in the log. The Fields directive specifies the data recorded in the fields of each entry.  
  
The following is an example of a record in the extended log format that was produced by the Microsoft Internet Information Server (IIS):  
#Software: Microsoft Internet Information Server 4.0  
#Version: 1.0  
#Date: 1998-11-19 22:48:39  
#Fields: date time c-ip cs-username s-ip cs-method cs-uri-stem cs-uri-query sc-status sc-bytes cs-bytes time-taken cs-version cs(User-Agent) cs(Cookie) cs(Referrer)  
  
1998-11-19 22:48:39 206.175.82.5 - 208.201.133.173 GET /global/images/navlineboards.gif - 200 540 324 157 HTTP/1.0 Mozilla/4.0+(compatible;+MSIE+4.01;+Windows+95) USERID=CustomerA;+IMPID=01234 http://yourturn.rollingstone.com/webx?98@@webx1.html  
  
Notes:  
  
Cookies in the W3C format (in contrast to the Combined format), are not quoted, and multiple cookies are delimited by + characters. Also note that Web Site Analyzer parses the + sign in cookies and user-agents by treating them as space characters. (It also strips the spaces at the beginning and end of each individual key and value.) This is important to keep in mind especially when creating filters. (See the online help in the filters window for more information.)  
  
Certain report elements give you a choice of a Time Taken measure. This measure is only relevant when analyzing data in the IIS format. (If you select Time Taken to report on data that is not in IIS format, no data will be returned.)  
  
SunTM ONE Web Server(iPlanet)  
      A log file in the One Web Server (iPlanet) format contains a sequence of lines containing ASCII characters. Each line may contain either a directive or an entry.  
  
Entries consist of a sequence of fields relating to a single HTTP transaction. Fields are separated by white space. If a field is unused in a particular entry dash "-" marks the omitted field.  
  
Directives, which appear at the beginning of the log, define the information and format contained in the entries.  
  
The following is an example of a record in the SunTM ONE Web Server (iPlanet) log format:  
  
  
format=%Ses->client.ip% - %Req->vars.auth-user% [%SYSDATE%]  
"%Req->reqpb.clf-request%" %Req->srvhdrs.clf-status%  
%Req->srvhdrs.content-length%  
10.101.128.66 - - [03/Apr/2002:00:42:11 -0500] "GET /multimedia/flash/dis.swf HTTP/1.1" 304  
  
Note that there are a large number of tokens that can be written to an IPlanet log file. Some of these tokens can cause duplicate data to be logged. Consequently, to improve performance, you should configure your server to eliminate such tokens.  
  
For example, in the following log sample, the directive (shown in red) duplicates request information (shown in blue) in the log and can thus be eliminated:  
  
format=%Ses->client.ip% - %Req->vars.auth-user% [%SYSDATE%]  
"%Req->reqpb.clf-request%"  
%Req->srvhdrs.clf-status% %Req->srvhdrs.content-length%  
"%Req->headers.referer%"  
"%Req->headers.user-agent%"  
%Req->reqpb.method%  
%Req->reqpb.uri% %Req->reqpb.query%  
"%Req->reqpb.protocol%"  
  
216.142.233.73 - - [30/Nov/2001:00:03:08 -0500]  
"GET /NASApp/KornPublic/KornPageServlet?pageId=Colors\_Detail\_ColorDetailLunarPearl&langId=701  
&surfaceId=602&roleId=801&rgnId=901 HTTP/1.1"  
200 - "http://www.Korn.com/NASApp/KornPublic/KornPageServlet  
?pageId=Colors\_Complete\_ZodiaqColorPalette  
&langId=701&surfaceId=602&roleId=801&rgnId=901"  
"Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)  
"GET /NASApp/KornPublic/KornPageServlet pageId=Colors\_Detail\_ColorDetailLunarPearl&langId=701  
&surfaceId=602&roleId=801&rgnId=901 "HTTP/1.1"  
  
IBM Tivoli Access Manager WebSEAL  
      IBMR TivoliR Access Manager WebSEAL is a resource security manager for Web-based resources.  
  
To import data from IBM Tivoli Access Manager WebSEAL, you must configure WebSEAL to create logs in the NCSA Combined log format.  
  
For information on how to do this, refer to the IBM Tivoli Access Manager WebSEAL Administrator's guide and search on the keyword logcfg  
WebSphere Application Server (WAS) Log Format  
      WebSphere Application Server (WAS) version 4.0 provides an API for application level logging called Analytic Logging Service (ALS). You can use call this API from your own Web applications. The IBM WebSphere Personalization server uses this API to log data.  
  
This API supports three logging methods: HTTP, database, and file. The record format for logged data differs slightly based on the logging method used; however, each log record contains the following five fields:  
  
appid reqid httpdata cookie appdata  
  
Example: <custCareID> <01010123-1234> <-> <-> <target=TP600E;crosssells=TP770,TP380;ads=pIII>  
The following is a description of these fields:  
  
appid <custCareID> in the example)  
The Application ID of the record. Appid allows multiple applications to log to the same medium.  
  
reqid <01010123-1234> in the example)  
The HTTP request correlation ID. The reqid is used for correlation of the JSP request to serve an HTTP request. Analysis tools can use this field as a key to identify records that belong to an HTTP request.  
  
httpdata <-> in the example)  
The NCSA Combined log. The fields in the log are host rfc931 username date request statuscode bytes referrer user\_agent and cookie.  
  
cookie <-> in the example)  
The cookie information.  
  
appdata <target=TP600E;crosssells=TP770,TP380;ads=pIII> in the example)  
The application data. The appdata field can contain any application level data. The format of the APPDATA field is a list of semicolon-separated key=values pairs, where key represents the semantic label of the values, and the values is a list of comma-separated tags or product identifiers. If a comma is part of the value, quotes are used to enclose the value.  
  
  
Custom Formats  
      If the server you are using produces a non-standard log format, or if you want to analyze a FTP log, you can use Web Site Analyzer to create a custom log format.  
  
The following information describes the field formats for a custom log file format. You can use this information to create the format in the Web Site Analyzer Global Settings window.  
  
You can also create your own custom parser to analyze the custom log format that you create.  
  
Custom Log Field Formats (field information defined by user) :  
  
This log file format does not contain a field directive line. Therefore, you should provide the details of the field definitions. The fields should be defined in the order that the fields occur in the log record.  
Important: The log file must contain resource, IP address, date, and time fields for it to be analyzed.  
  
For each field, you should provide the following:  
  
Field Type  
Type of the field. There are several pre-defined field types. The parser will ignore any field with the FID\_IGNORE field type. You can use your own field type id provided the value is greater than 100.  
  
Begin Delimiter  
The delimiter that identifies the beginning of the field. The delimiter is not the part of the field. If the field begins right after the end delimiter of the previous field, the value for the Begin Delimiter must be null. You can not use the end delimiter of the previous field as the begin delimiter.  
  
End Delimiter  
The delimiter that identify the end of the field. The delimiter is not the part of the field. If the field is the rest of the line, the value of the End Delimiter must be null.  
  
Parser Class Name  
The class name of the custom field parser. The default field parser is used if you specify DEFAULT. The default parser for the user defined input field type is WsaKeyValuePairParser. (See creating your own parsers for more information.)  
  
Comment Line Token  
The token to identify the comment line. For a custom log format, it can be"#" or any string that you want. The only condition is that it be different from the first field beginning delimiter.  
  
Date Format  
The format of the date. Use standard symbols to define the date format. For a custom log format it is "yyyy-MM-dd "  
  
where:  
  
Symbol     Meaning     Format     Example  
y     year     (Number)     1996  
M     month in year     (Text or Number)     July or 07  
d     day in month     (Number)     10  
  
Time Format  
The format of the date. Use standard symbols to define the date format. For a custom log format it is "HH:mm:ss "  
  
where:  
  
Symbol    Meaning    Format    Example  
h     hour in am/pm (1~12)     (Number)     12  
H     hour in day (0~23)     (Number)     0  
m     minute in hour     (Number)     30  
s     second in minute     (Number)     55  
a     am/pm marker     (Text)     PM  
  
NULL Token  
The token to use for a field with a null value. For a custom log format it is "-"

## 부록 2 polliwog를 사용한 웹로그 분석

**개요**

awstats를 사용하여 웹로그를 분석해 본적이 있다면 polliwog는 awstats의 자바 버전이라고 생각하면 이해하기가 쉬울 듯 하다.

**설치**

먼저 아래의 url에서 다운로드 한다.

http://sourceforge.net/projects/polliwog/files/polliwog/

이때 아래와 같은 세가지 파일이 있는데 여기에서는 polliwog-bin-stable-0.7.tar.gz를 다운로드했다.

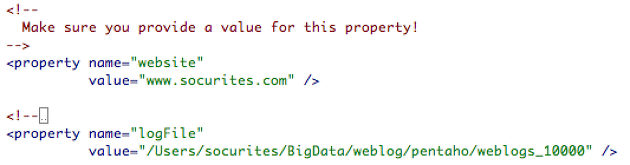


현재는 stable-0.7 버전이 안정 버전인 듯 하며, 2009년 이후로 개발되지는 않는 듯 하다.

먼저 압축을 푼다. 그런 후 build.xml 파일을 열어서 아래의 두가지 속성을 수정해야 한다.

* website  
  분석할 대상 사이트의 URL. 분석한 결과를 HTML 파일로 만들어주는데, 이처럼 자동생성될 HTML 파일에서 사용할 사이트의 URL을 적당히 입력한다.
* logFile  
  분석할 로그 파일의 위치. 파일이 포함된 디렉토리가 아니라, 파일 하나를 지정해야 한다.

여기에서는 아래와 같이 수정했다.



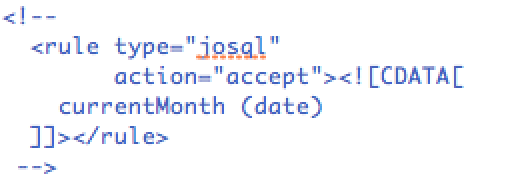
**실행**

ant를 사용하여 build.xml을 실행한다.

$ ant

그러면 사용한 로그 파일에 따라 예외가 날 수 있다. 기본 설정에서는 현재 날짜의 로그만 분석하도록 필터링이 설정되어 있다.

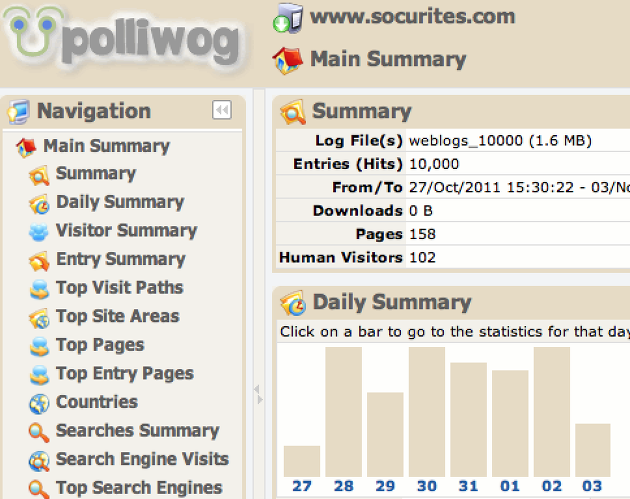
따라서 이전 날짜의 로그를 분석하려면 필터링 규칙을 제거해야 한다. data/hit-filter-config.xml 파일에서 아래 부분을 다음과 같이 주석처리 한다.



ant를 다시 실행하면 아래와 같이 분석 로그와, 분석된 결과를 저장한 html 파일 위치가 출력된다.

http://cfile5.uf.tistory.com/image/125AF4445105212211F56B

생성된 html 파일(monthly-summary.html)을 열어보면 아래와 같이 분석 정보를 볼 수 있다.



**[참고자료]**

* [polliwog 사이트](http://polliwog.sourceforge.net/)
* [polliwog Getting Started](http://polliwog.sourceforge.net/manual/getting-started.html)

## 부록 3. Awstats를 사용한 웹로그 분석

**개요**

AWStats는 웹서버, 스트리밍 서버, ftp 서버, 메일 서버 등에 대한 고급 통계 정보를 생성하는 강력하고 다양한 기능을 가진 무료 도구다. AWStats는 로그 분석기로써, CGI 방식으로 동작하거나 또는 커맨드 라인에서도 실행할 수 있다. AWStats는 로그에 포함되어 있는 정보를 최대한 추출하여 그래픽한 웹 페이지로 표시해 준다.

AWStats는 대량의 로그 파일을 자주 그리고 빠르게 처리하기 위해서 메타 데이터가 포함된 파일을 사용한다. AWStats는 메이저급 서버에서 생성하는 대다수의 로그 파일을 분석할 수 있다. 예를 들어 아파치 로그 파일(NCSA combined / XLF / ELF 로그 포맷 또는 common/CLF 로그 포맷), WebStar, IIS(W3C 로그 포맷) 뿐만 아니라 다수의 웹 서버, 프록시 서버, wap 서버, 스트리밍 서버, 메일 서버와 일부 ftp 서버의 로그도 분석할 수 있다.

AWStats는 GNU GPL을 따르는 무료 소프트웨어다. 널리 알려진 다른 도구와의 차이점이 궁금하다면 [여기](http://awstats.sourceforge.net/docs/awstats_compare.html)를 방문해 보라.

**설치하기**

**[참고자료]**

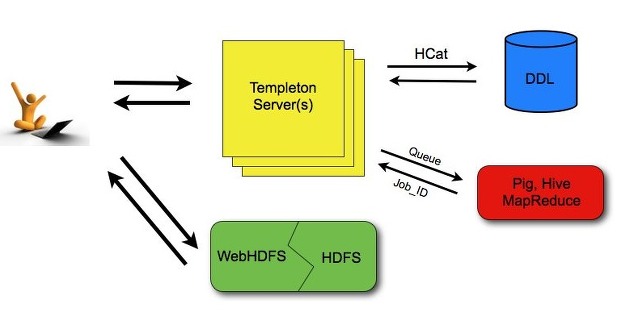
* [AWStats 사이트](http://awstats.sourceforge.net/)
* [AWStats Installation, Configuration and Reporting](http://awstats.sourceforge.net/docs/awstats_setup.html)

## 부록 4. [번역] 템플톤(Templeton)이란

원문 : http://people.apache.org/~thejas/templeton\_doc\_latest/index.html

**개요**

템플톤(Templeton)은 HCatalog 등 하둡 컴포넌트에 대해 Rest 형태의 웹 API를 제공한다. 아래 그림에서 보는 것과 같이 개발자는 응용 프로그램에서 HTTP 요청을 사용하여 하둡 맵리듀스, 피그, 하이브, HCatalog DDL에 접근할 수 있다. 템플톤에서 사용하는 데이터와 코드는 HDFS에 저장되어 있다. HCatalog DDL 명령문은 요청받은 즉시 실행된다. 반면 맵리듀스, 피그, 하이브 Job은 템플톤에 의해 큐에 저장되어 처리 상황을 모니터링하며, 필요에 따라 Job을 중지할 수도 있다. 개발자는 피그, 하이브, 맵리듀스 Job의 결과를 저장할 HDFS 경로를 템플톤에 알려주어야 한다.



**URL 포맷**

템플톤 자원은 아래와 같은 URL 포맷을 사용하여 접근할 수 있다.

http://yourserver/templeton/v1/resource

이때 *"yourserver"* 은 템플톤 서버 주소이고, "resource"는 템플톤 자원 명이다.

예를 들어 템플톤 서버가 현재 기동중인지 확인하려면 아래와 같은 URL로 접근할 수 있다.

http://www.myserver.com/templeton/v1/status

**보안**

현재 버전의 템플톤은 두가지 형태의 보안을 지원한다.

* 기본 인증 (별도의 인증을 하지 않는다)
* 커버로스(Kerberos) 인증

**표준 파라미터**

인증을 지원하기 위해 템플톤 자원은 모두 아래의 파라미터를 받아들인다.

* user.name : 문자열로 된 사용자 명. 기본 인증 방식을 사용할 때만 의미가 있다.
* SPNEGO 인증 : 커버로스 인증을 사용하는 경우

**보안 실패 응답 메시지**

필요한 user.name 파라미터가 요청에 없는 경우, 템플톤은 아래와 같은 에러 메시지를 반환한다.

{

  "error": "No user found.  Missing user.name parameter."

}

**WebHDFS와 코드 밀어넣기**

템플톤 자원으로 사용하는 데이터와 코드는 먼저 하둡에 저장되어 있어야 한다. 현재 버전의 템플톤에서는 WebHDFS와 같이 이러한 작업을 처리할 수 있는 기존의 웹 인터페이스를 통합하거나 대체하지는 않았다. (이러한 기능 통합은 아마 향후 릴리즈에 개발할 것이다.)  따라서 필요한 데이터나 코드를 HDFS에 저장해야 한다면, 편한 방법을 선택하라.

**에러 코드와 응답 메시지**

**템플톤 서버는 아래와 같은 HTTP 상태 코드를 반환한다.**

* **200 OK:** Success!
* **400 Bad Request:** The request was invalid.
* **401 Unauthorized:** Credentials were missing or incorrect.
* **404 Not Found:** The URI requested is invalid or the resource requested does not exist.
* **500 Internal Server Error:** We received an unexpected result.
* **503 Busy, please retry:** The server is busy.

템플톤은 데이터를 모두 JSON 형태로 반환한다. JSON 응답 메시지는 1MB로 크기가 제한된다. 만약 응답 메시지가 1MB보다 크다면 직접 반환하는 대신, HDFS에 저장해야 한다. 만약 HCatalog DDL 명령문의 결과가 1MB보다 크다면, 차라리 이에 대응하는 하이브 요청을 실행하는 편이 낫다.

**로그 파일**

템플톤은 기동될 때 세가지 로그 파일을 생성한다.

* templeton.log  
  log4j 로그. 응용 프로그램이 기록하는 메인 로그
* templeton-console.log  
  템플톤이 기동될 때 자바의 stdout으로 출력된 로그. "hcat.out"에 비교해서 사이즈가 작다.
* templeton-console-error.log  
  템플톤이 기동될 때 자바의 stderr으로 출력된 로그. "hcat.err"과 사이즈가 비슷하다.

템플톤의 log4j.properties 파일의 templeton.log.dir 속성을 사용하여 이들 로그 파일의 위치를 설정할 수 있다. log4j.properties 파일은 템플톤 서버 기동 스크립트에서 설정된다.

**프로젝트 명**

The Templeton project is named after the a character in the award-winning children's novel Charlotte's Web, by E. B. White. The novel's protagonist is a pig named Wilber. Templeton is a rat who helps Wilber by running errands and making deliveries.

## 부록 5. ANTLR을 사용한 웹로그 파서 만들기

**ANTLR을 사용한 웹로그 파서 만들기**

http://i1.daumcdn.net/icon/editor/p_zip_s.gif weblogparser.zip

**컨텍스트**

웹로그의 경우 아래와 같이 그 형식이 다양하다.

**[iis의 경우]**

*2012-03-28 23:09:59 112.106.12.140 GET /admin/module-builtin.xml - 443 - 83.170.77.64 Mozilla/4.0+(compatible;+MSIE+5.01;+Windows+NT+5.0) 404 0 2 390*

**[sun one의 경우]**

*[30/Mar/2012:00:03:18 +0900] "POST /extra/popup\_member.jsp?first\_menu=member&second\_menu=find&third\_menu=find\_popidsearch HTTP/1.1" 302 439*

문자열을 파싱할 때 대개의 경우 구분자 주도 변환 기법을 사용한다. 즉, 파싱하려는 문자열에서 사용하지 않는 특수한 문자를 구분자로 선택한 후, 그 구분자를 사용해 문자열을 나누는 방식이다. 위와 같은 로그라면 공백을 사용해 문자열을 분할할 수 있지 않을까라는 생각이 먼저 떠오른다.

하지만 생각과는 달리, 구분자 주도 변환 기법은 이 경우에 그리 효과가 없다. sun one의 로그를 보면 알겠지만 URL 부분의 경우 쌍따옴표로 묶인 부분을 하나의 토큰으로 분할해야하지만, 쌍따옴표 안에 공백이 이미 사용되고 있기 때문이다.

구분자 대신 사용할 수 있는 방법으로 정규 표현식을 고려해볼 수 있다. 하지만 정규 표현식을 사용하게 되면, 정규 표현식 자체가 복잡할뿐만 아니라, 정규식으로 추출한 토큰은 그 순서에 따라 의미가 달라지게된다. 예를 들어 iis의 경우에는 두 번째 토큰이 IP인데 반해, sun one의 경우에는 URL과 관련된 정보다.

결국 웹로그 파서를 만드는 경우, 구분자 주도 변환 기법이나 정규식을 이용해 파서를 직접 만드는 방식 모두 효과가 없다.

**해결책**

문법을 작성하고, 문법을 기반으로 파서를 생성하는 구문 주도 변환 기법을 사용하고자 한다. 파서 생성기로는 널리 알려진 ANTLR을 사용한다. 그리고 파서의 결과로는 트리를 생성한다. 파서에서 시맨틱 모델을 바로 생성할 수도 있지만, 추상 구문 트리(AST)를 만든 후, 트리를 다시 시맨틱 모델로 변환하는 방식이 더 효과적이다. 웹로그를 파싱하는 경우 시맨틱 모델은 단순히 웹로그 도메인 객체다.

**Step By Step**

**1. 설치하기**

[ANTRL 사이트](http://www.antlr.org/works/index.html)에서 ANTLRWorks를 다운로드한다. 현재 최신 버전은 1.4.3이다.

Window의 경우, 다운로드 받은 디렉토리로 이동해서 아래의 명령어를 실행한다.

java -Xmx512m -jar antlrworks-1.4.3.jar

Window 이외의 플랫폼을 사용중이라면, [How to run ANTLRWorks](http://www.antlr.org/works/help/tutorial/howtorun.html)를 참고한다.

**2. 문법 만들기**

아래와 같이 문법 파일을 작성한다. ANTLR에서는 렉싱 규칙과, 파싱 규칙을 하나의 문법에서 작성한다.

# 문법 파일...

grammar Weblog;  
  
options {  
  output=AST;  
  
}  
  
tokens { ATTRIBUTE; ATTRIBUTE\_LIST;  
}  
  
log            : attr+ ;    
attr            : ID ;  
  
ID        : (LETTER | DIGIT |'-'|'('|')'|'/'|'\_'|'.'|':'|';'|'+')+ | '[' (.)\* ']' | '"' (.)\* '"' ;  
  
NEWLINE:'\r'? '\n' ;  
WS  :   (' '|'\t')+ {skip();} ;  
  
fragment LETTER: LOWER | UPPER;  
fragment LOWER: 'a'..'z';  
fragment UPPER: 'A'..'Z';  
fragment DIGIT: '0'..'9';

위와 같이 문법파일을 작성한 후, Weblog.g라는 이름으로 저장한다. 이때 최상위의 문법명(Weblog)와 파일명은 같아야 한다.

**4. 렉싱 문법 테스트하기**

작성한 렉싱 문법을 테스트한다. 하단의 Interpreter탭을 연다. 왼쪽의 텍스트 영역에 테스트할 입력 텍스트를 붙여넣는다. 여기에서는 아래의 웹로그를 테스트한다.

[30/Mar/2012:00:03:18 +0900] "POST /extra/popup\_member.jsp?first\_menu=member&second\_menu=find&third\_menu=find\_popidsearch HTTP/1.1" 302 439

그리고나서, 명령버튼 영역에서 실행버튼(▶)을 클릭하면 아래와 같은 파스 트리를 볼 수 있다.

**5. 트리 생성 구문 추가하기**

렉싱 규칙에 대한 테스트가 끝나면, 이제 트리를 생성하는 구문을 문법에 추가한다. ANTLR에서는 트리를 쉽게 생성할 수 있는 특수한 구문을 제공한다. 아래와 같이 문법을 수정한다.

# 문법 파일...

grammar Weblog;

...  
log            : attr+ **-> ^(ATTRIBUTE\_LIST attr+)** ;    
attr            : ID **-> ^(ATTRIBUTE ID)** ;  
...

**6. 렉서 코드와 파서 코드 생성하기**

문법파일 작성이 완료되면, 아래의 명령어를 실행해서 렉서와 파서를 생성한다.

java -cp .\antlrworks-1.4.3.jar org.antlr.Tool Weblog.g

코드가 성공적으로 생성되면, 하위 폴더에 Weblog.tokens, WeblogLexer.java, WeblogParser.javar, 이렇게 세 파일이 생성된다.

또는 Generate 메뉴에서 [Generate Code] 메뉴를 선택한다. 이경우에는 문법파일이 위치한 디렉토리의 하위 디렉토리인 output에 생성된다. 디버깅 툴에서는 생성된 코드가 output 디렉토리에 있다고 가정하므로, 메뉴를 통해서 코드를 생성할 것을 추천한다.

**7. 파싱 규칙 테스트하기**

트리 생성 구문까지 작성이 완료되면, 파싱 규칙을 테스트한다. 문법 파일을 수정했으므로, 코드를 새로 생성한다.

코드 생성이 끝나면, Run 메뉴에서 [Debug...] 메뉴를 선택한다.

텍스트  영역에, 테스트하려는 로그 텍스트를 붙여넣은 후, OK 버튼을 누른다. 그리고 나서 [Go to End] 버튼을 클릭하면, 아래와 같이 파스 트리가 간소화시킨 AST가 정상적으로 만들어짐을 확인할 수 있다. (때에 따라 원격 서버에 접속할 수 없다는 메시지가 뜨는 경우가 있는데, 이 경우에는 ANTLRWorks를 재기동한다).

**8. 생성할 코드에 속성 부여하기**

아무런 속성을 지정하지 않을 경우, 렉서와 파서는 디폴트 패키지에 생성된다. 따라서 패키지를 설정하거나, 옵션을 주어 생성할 소스코드의 속성을 수정한다.

# 문법 파일...

grammar Weblog;  
  
**options {  
  output=AST;  
}**  
...  
**@header {  
  package com.socurites.weblog.parser;  
}  
  
@lexer::header {  
  package com.socurites.weblog.lexer;  
}**  
...

문법이 변경되었으므로, 코드를 다시 생성한다.

**9. 시맨틱 모델 클래스 만들기**

여기에서는 단순히 웹로그를 저장할 도메인 클래스 하나면 충분하다. 여기에서는 별도의 도메인 클래스를 만들지 않고 HashMap<String, String>에 저장한다.

**9. 로더 프로그램 만들기**

이제 렉서와 파서를 실행할 로더 프로그램을 만든다. 먼저 아래와 같이 이클립스에서 자바 프로젝트를 하나 생성한다. 그리고 다운로드받은 antlrworks-1.4.3.jar 파일을 라이브러리에 추가한다. 여기에서는 메이븐 프로젝트를 만들었고, 의존성 라이브러리에는 antlr-runtime과 junit4를 추가했다.

로더 클래스를 만들기 전, 테스트 케이스를 작성한다. 테스트케이스는 단순하다. 테스트를 실행하기 전, 로그데이터를 담고있는 파일을 읽어서 Reader 객체에 저장한다.

Reader 객체를 로더 객체를 생성하면서 생성자의 인자로 전달한다.

package com.socurites.weblog.loader;  
  
import static org.hamcrest.CoreMatchers.is;  
import static org.hamcrest.CoreMatchers.notNullValue;  
import static org.junit.Assert.assertThat;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.InputStreamReader;  
import java.io.Reader;  
import java.util.Map;  
  
import org.junit.Before;  
import org.junit.Test;  
  
  
public class **WeblogLoaderTest**{  
    Reader reader;  
      
    @Before  
    public void init() throws IOException {  
        InputStream is4Sample = this.getClass().getResourceAsStream("/sample.properties");  
        reader = new InputStreamReader(is4Sample);  
    }  
      
    @Test  
    public void run() {  
        WeblogLoader loader = new WeblogLoader(reader);  
        loader.run();  
        Map<String, String> weblog = loader.getWeblog();  
          
        assertThat(weblog, is(notNullValue()));  
        assertThat(weblog.get("0"), is("203.226.192.157"));  
          
    }  
}

로더 클래스에서 가장 핵심이 되는 메서드는 run()으로, AST를 로드한 후 시맨틱 모델을 만드는 역할을 한다.

package com.socurites.weblog.loader;  
  
import java.io.IOException;  
import java.io.Reader;  
import java.util.HashMap;  
import java.util.Map;  
  
import org.antlr.runtime.ANTLRReaderStream;  
import org.antlr.runtime.CommonTokenStream;  
import org.antlr.runtime.RecognitionException;  
import org.antlr.runtime.tree.CommonTree;  
  
import com.socurites.weblog.lexer.WeblogLexer;  
import com.socurites.weblog.parser.WeblogParser;  
  
public class **WeblogLoader**{  
    private Reader input;  
    private CommonTree ast;  
    private Map<String, String> **weblog**;  
  
    public WeblogLoader(Reader input) {  
        this.input = input;  
    }  
      
    public void **run**() {  
        loadAST();  
        createWeblog();  
    }  
  
    private void **loadAST**() {  
        try {  
            WeblogLexer lexer = new WeblogLexer(new ANTLRReaderStream(input));  
            WeblogParser parser = new WeblogParser(new CommonTokenStream(lexer));  
              
            ast = (CommonTree) parser.log().getTree();  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        } catch (RecognitionException e ) {  
            throw new RuntimeException(e);  
        }          
    }  
  
    private void **createWeblog**() {  
        weblog = new HashMap<String, String>();  
        int i = 0;  
        for ( Object obj : ast.getChildren() ) {  
            CommonTree node = (CommonTree) obj;  
            weblog.put("" + i++, node.getChild(0).getText());  
        }  
    }  
  
    public Map<String, String> getWeblog() {  
        return weblog;  
    }  
}

로그데이터를 담고 있는 샘플 파일을 만든다.

# sample.properties

203.226.192.157 - - [30/Mar/2012:17:38:42 +0900] "GET /upload/adm/banner/main/tmp/bottom\_P\_L\_2.jpg HTTP/1.1" 200 20048 "http://www.samsungsdi.co.kr/front/swf/Main.swf"

테스트케이스를 실행하면 성공했음을 알 수 있다.

**추가할 사항**

AST에서 자식 노드를 가져올 때 그 순서는 웹로그에 따라 다르다. 따라서 가져온 자식 노드가 어떤 데이터인지 알려면, 각 웹로그 유형별로 로그의 타입정보를 저장해야 한다. 그리고 나서 런타임에 로그 타입 정보를 읽어서, 몇 번째 노드가 무슨 데이터인지 인식하도록 프로그램을 개선해야 한다.

**참고자료**

* [ANTLR 사이트](http://www.antlr.org/works/index.html)
* [How to run ANTLRWorks](http://www.antlr.org/works/help/tutorial/howtorun.html)
* [Using ANTLRWorks with Java.g](http://openjdk.java.net/projects/compiler-grammar/antlrworks/index.html)
* [Building ANTLR Projects with Maven](http://www.antlr.org/wiki/display/ANTLR3/Building+ANTLR+Projects+with+Maven)
* [ANTLR FAQ - Getting Started](http://www.antlr.org/wiki/display/ANTLR3/FAQ+-+Getting+Started)
* [ANTLR Expression evaluator](http://www.antlr.org/wiki/display/ANTLR3/Expression+evaluator)

## 부록 6. 하둡 분산 파일 시스템을 기반으로 색인하고 검색하기

**하둡 분산 파일 시스템을 기반으로 색인하고 검색하기**

**- 인터넷에서 바늘 찾기**-

원문: http://www.drdobbs.com/article/print?articleID=226300241

July 29, 2010

*Kashyap Santoki는  Infosys Technologies Limited에서 일한다. KashyapChimanlal\_S@infosys.com로 연락할 수 있다.*

오늘날에는 정보가 넘치고 있다. 지역적으로 흩어져 있는 거대한 량의 정보는 계속해서 증가하고 있으며, 이러한 정보로부터 의미있는 결과를 얻어내려면 빠르게 파싱할 수 있는 시스템을 필요로 한다. 분산 데이터를 검색할 수 있는 색인을 만들 수 있다면, 파싱작업을 하는데 큰 도움이 된다. 이 글에서는 루씬(Lucene)와 자바를 사용해서 데이터를 색인하고 검색하는 기본적인 방법을 설명한다. 또한 RAMDirectory를 사용해서 색인하고 검색하는 방법, HDFS에 저장된 데이터에 대해 색인을 생성하는 방법, 이렇게 생성된 색인을 검색하는 방법을 설명한다. 마이크로소프트 윈도우 XP SP3을 플랫폼으로 사용하며, 개발환경으로는 자바 1.6, 이클립스 3.4.2, 루씬 2.4.0, 하둡 0.19.1으로 구성된다.

색인과 검색 작업을 위해서 하둡을 사용했다. 아파치 하둡 프로젝트는 안전하며, 확장 가능한 분산 컴퓨팅을 지원하는 오픈소스 소프트웨어다. 하둡 분산 파일 시스템(HDFS)는 원거리 네트워크상에서 파일을저장하고 공유하기 위한 목적으로 만들어졌다. HDFS는 일반적인 하드웨어상에서 동작할수 있도록 만들어졌다. 또한 장애 복구와 자원 관리 기능을 제공하며, 무엇보다도 응용 데이터 접근에 대한 높은 가용성을 지원한다.

**로컬 파일 시스템에 색인 만들기**

우선 로컬 파일 시스템에 저장된 데이터에 대한 색인을 만들어야 한다. 이클립스를 사용해서 먼저 프로젝트를 하나 생성하고, 클래스를 하나 만든다. 그리고 필요한 JAR 파일을 모두 프로젝트에 추가한다. 아래와 같이 일반적인 웹서버에서 생성하는 로그 파일을 데이터  예제로 사용하려고 한다.

2010-04-21 02:24:01 GET /blank 200 120

위 데이터는 아래와 같은 필드로 구성된다.

* 2010-04-21 -- 날짜 필드
* 02:24:01 -- 시간 필드
* GET -- 메서드 필드 (GET 또는 POST) -- 앞으로는 "cs-method"라는 용어로 사용하겠다
* /blank -- 요청 URL 필드 -- 앞으로는 "cs-uri"라는 용어로 부르겠다
* 200 -- 요청에 대한 상태 코드 -- "sc-status"라고 부르겠다
* 120 -- 처리 시간 필드 (요청을 처리하는데 걸린 시간)

샘플 파일은 "E:\DataFile" 디렉토리에 "Test.txt"라는 이름으로 저장되어 있다고 가정한다. 샘플 파일의 내용ㅇ은 다음과 같다.

2010-04-21 02:24:01 GET /blank 200 120

2010-04-21 02:24:01 GET /US/registrationFrame 200 605

2010-04-21 02:24:02 GET /US/kids/boys 200 785

2010-04-21 02:24:02 POST /blank 304 56

2010-04-21 02:24:04 GET /blank 304 233

2010-04-21 02:24:04 GET /blank 500 567

2010-04-21 02:24:04 GET /blank 200 897

2010-04-21 02:24:04 POST /blank 200 567

2010-04-21 02:24:05 GET /US/search 200 658

2010-04-21 02:24:05 POST /US/shop 200 768

2010-04-21 02:24:05 GET /blank 200 347

"Test.txt" 파일에 저장된 데이터에 대해 색인을 만들어서, 색인을 로컬 파일 시스템에 저장하려고 한다. 아래의 자바 코드를 사용하면 이 작업을 할 수 있다. (각 코드가 어떤 작업을 처리하는지 주석을 상세히 달아 두었다).

// IndexWriter 객체를 생성한다. 이때 색인 파일을 저장할 경로를 인자로 전달한다.  
**IndexWriter** indexWriter = new IndexWriter("E://DataFile/IndexFiles", new **StandardAnalyzer**(), true);  
              
// BufferedReader 객체를 생성한다. 이때 색인하려고 하는 데이터가 저장된 파일 경로를 인자로 전달한다.

**BufferedReader**reader= new BufferedReader(new FileReader("E://DataFile/Test.txt"));

String row=null;  
          
// 데이터 파일에서 각 라인을 읽는다.

while ((row=reader.readLine())!= null) {  
    // 한 행을 분할하여 각 필드를 구한 후, 배열에 저장한다. 데이터 파일에서 사용한 구분자는 공백 문자다  
    String Arow[] = row.split(" ");

    // 각 행에 대해 document 객체를 생성한 후, 해당 document 객체에 필드를 데이터로 추가한다.  
    org.apache.lucene.document.**Document** document = new org.apache.lucene.document.Document();

**document.add**(new **Field**("date",Arow[0],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field("time",Arow[1],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("cs-method",Arow[2],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("cs-uri",Arow[3],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("sc-status",Arow[4],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("time-taken",Arow[5],Field.Store.YES,Field.Index.ANALYZED));  
                  
    // document 객체를 색인 파일에 추가한다.  
    **indexWriter**.**addDocument**(document);  
}          
**indexWriter.optimize**();  
**indexWriter.close**();  
reader.close();

**로컬 색인 파일 검색하기**

이제 생성된 색인 파일에서 데이터를 검색할 수 있다. 기본적으로 검색은 "필드" 데이터 기반으로 수행된다. 루씬 검색 엔진에서 지원하는 다양한 검색 기능을 사용해서 검색할 수 있다. 또한 특정 필드 하나만을 사용할 수도 있고, 필드를 조합해서 검색할 수도 있다. 아래의 자바 코드는 색인을 검색하는 코드다.

// Searcher객체를 생성한다. 이때 인덱스 파일이 저장된 경로를 인자로 전달한다.

**Searcher**searcher = new **IndexSearcher**("E://DataFile/IndexFiles");  
**Analyzer**analyzer = new **StandardAnalyzer**();

// 현재 색인 파일에 저장된 문서 또는 항목의 총 갯술르 출력한다.  
System.out.println("Total Documents = "+searcher.**maxDoc**()) ;

// QueryParser 객체를 생성한다. 이때 검색을 적용할 필드명을 인자로 전달한다.

**QueryParser** parser = new QueryParser("cs-uri", analyzer);

//Query 객체를 생성한다. 이때 검색할 텍스트를 인자로 전달한다.  
**Query** query = parser.parse("/blank");

// 아래의 문장을 실행하면 색인 파일에서 검색을 수행하게 된다.             
**Hits** hits = searcher.**search**(query);

// 검색 질의문에 매칭하는 문서 또는 항목의 갯수를 출력한다.              
System.out.println("Number of matching documents = "+ **hits.length()**);

// 검색 조건에 매칭하는 문서(똔느 파일의 행)을 출력한다.

for (int i = 0; i < hits.length(); i++) {  
    **Document** doc = **hits.doc(i)**;  
    System.out.println(doc.**get**("date")+" "+ doc.get("time")+ " "+  
    doc.get("cs-method")+ " "+ doc.get("cs-uri")+ " "+ doc.get("sc-status")+ " "+ doc.get("time-taken"));  
}

이 예제에서는 **cs-uri**필드에 대해, 해당 **cs-uri**필드에  **/blank**가 포함된 텍스트를 검색했다. 따라서 검색 코드가 실행되면 **cs-uri**필드에  **/blank**가 포함된 모든 문서(또는 행)이 결과로 출력된다. 출력은 아래와 같다.

Total Documents = 11  
Number of matching documents = 7  
2010-04-21 02:24:01 GET /blank 200 120  
2010-04-21 02:24:02 POST /blank 304 56  
2010-04-21 02:24:04 GET /blank 304 233  
2010-04-21 02:24:04 GET /blank 500 567  
2010-04-21 02:24:04 GET /blank 200 897  
2010-04-21 02:24:04 POST /blank 200 567  
2010-04-21 02:24:05 GET /blank 200 347

**HDFS에서 메모리 기반으로 색인하기**

이제 데이터가 HDFS와 같은 분산 파일 시스템에 저장된 경우를 생각해보자. 이처럼 데이터가 분산된 경우에는 앞에서 설명한 색인 파일을 바로 생성하는 코드는 제대로 동작하지 않는다. 따라서 색인 파일을 생성하기 전에 먼저 몇가지 작업을 우선 처리해야 한다. 예를 들어 HDFS에 저장된 데이터를 로컬 파일 시스템에 복사하는 작업, 로컬 파일 시스템에 해당 데이터에 대한 색인을 생성하는 작업, 마지막으로 색인 파일을 다시 HDFS에 저장하는 단계를 거쳐야 한다. 검색할 때도 비슷한 과정을 따라 처리해야 한다. 하지만 이러한 작업은 시간이 오래 걸릴뿐더러, 그리 좋은 방식이 아니다. 대신에 데이터가 저장되어 있는 HDFS 노드의 메모리를 사용해서 데이터를 색인하고 검색해보자.

앞에서 사용했던 데이터 파일인 "Test.txt"가 이제 HDFS의 "/DataFile/Test.txt" 경로에 저장되어 있다고 가정하자. 그리고 생성된 색인 파일을 저장할 디렉토리를 HDFS의 "/IndexFiles"에 생성하자. 아래의 자바 코드는 HDFS에 저장된 파일에 대해 색인 파일을 메모리에 생성하는 코드다.

// 인덱스 파일을 생성할 경로

String Index\_DIR="/IndexFiles/";

// 데이터 파일이 저장된 경로

String File\_DIR="/DataFile/test.txt";

// HDFS에 접근하기 위한 FileSystem 객체를 생성한다.

Configuration config = new Configuration();  
config.set("fs.default.name","hdfs://127.0.0.1:9000/");  
FileSystem dfs = FileSystem.get(config);

// 색인을 메모리에 생성할 RAMDirectory (메모리) 객체를 생성한다.

**RAMDirectory** rdir = new RAMDirectory();

// RAMDirectory 객체에 대한 IndexWriter 객체를 생성한다.  
**IndexWriter**indexWriter = new IndexWriter (rdir, new **StandardAnalyzer**(), true);

// FSDataInputStream 객체를 생성한다. 이 객체를 사용해서 HDFS에 저장된 "Test.txt" 파일을 읽어들인다.  
FSDataInputStream filereader = dfs.open(new Path(dfs.getWorkingDirectory()+ File\_DIR));

String row=null;

// 파일에서 한 행씩 읽어들인다.

while ((row=reader.readLine())!=null) {

    // 한 행을 분할하여 각 필드를 구한 후, 배열에 저장한다. 데이터 파일에서 사용한 구분자는 공백 문자다.

    String Arow[]=row.split(" ");  
                  
    // 각 행에 대해 document 객체를 생성한 후, 해당 document 객체에 필드를 데이터로 추가한다.  
    org.apache.lucene.document.**Document**document = new org.apache.lucene.document.Document();  
                  
    **document.add**(new **Field**("date",Arow[0],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field("time",Arow[1],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("cs-method",Arow[2],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("cs-uri",Arow[3],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("sc-status",Arow[4],Field.Store.YES,Field.Index.ANALYZED));  
    **document.add**(new Field ("time-taken",Arow[5],Field.Store.YES,Field.Index.ANALYZED));  
                  
    // document 객체를 색인 파일에 추가한다.  
    **indexWriter**.**addDocument**(document);  
}

indexWriter.optimize();  
indexWriter.close();  
reader.close();

보다시피 "Test.txt" 파일은 HDFS에 저장되어 있으며, 이 파일에 대한 색인을 메모리에 생성했다. 색인 파일을 HDFS 디렉토리에 저장하려면.

// 메모리에 저장된 파일을 배열로 얻기

String fileList[]=rdir.**list**();

// 메모리로부터 인덱스를 읽어서 HDFS에 저장한다.  
for (int i = 0; I < fileList.length; i++) {  
    **IndexInput**indxfile = rdir.**openInput**(fileList[i].trim());  
    long len = indxfile.length();  
    int len1 = (int) len;

    // 파일로부터 바이트 배열로 데이터를 읽어들인다.

    byte[] bytarr = new byte[len1];  
    indxfile.**readBytes**(bytarr, 0, len1);

    // HDFS 디렉토리에 해당 인덱스 파일명을 가진 파일을 생성한다.              
    Path src = new Path(dfs.getWorkingDirectory()+Index\_DIR+ fileList[i].trim());  
    dfs.createNewFile(src);

    // 바이트 배열로부터 HDFS로 데이터를 쓴다.

    FSDataOutputStream fs = dfs.create(new Path(dfs.getWorkingDirectory()+Index\_DIR+fileList[i].trim()),true);  
    fs.write(bytarr);  
    fs.close();  
}  
dfs.closeAll();

이 작업을 통해 데이터 파일인 "Test.txt"에 필요한 색인 파일을 생성한 후, 생성된 색인 파일을 HDFS 디렉토리에 저장했다.

**HDFS에서 메모리 기반으로 검색하기**

이제 HDFS에 저장된 색인 파일에 대해 검색을 할 수 있다. 검색을 하려면 우선 HDFS에 저장된 색인 파일을 메모리로 올려야 한다. 아래의 코드를 보자.

// HDFS에 접근하기 위한 FileSystem 객체를 생성한다.  
Configuration config = new Configuration();  
config.set("fs.default.name","hdfs://127.0.0.1:9000/");  
FileSystem dfs = FileSystem.get(config);  
  
// 색인을 메모리에 생성할 RAMDirectory (메모리) 객체를 생성한다.  
RAMDirectory rdir = new **RAMDirectory**();  
  
// 해당 디렉토리에 저장된 색인의 목록을 얻어서 배열로 저장한다.  
Path pth = new Path(dfs.getWorkingDirectory()+Index\_DIR);  
FileSystemDirectory fsdir = new FileSystemDirectory(dfs,pth,false,config);  
String filelst[] = fsdir.list();  
  
FSDataInputStream filereader = null;  
for (int i = 0; i<filelst.length; i++) {  
    // HDFS 디렉토리에 저장된 색인 파일로부터 데이터를 filereader로 읽어들인다.

    filereader = dfs.open(new Path(dfs.getWorkingDirectory()+Index\_DIR+filelst[i]));

    int size = filereader.available();

    // 파일로부터 배열로 데이터를 읽어들인다.

    byte[] bytarr = new byte[size];

    filereader.read(bytarr, 0, size);

    // RAMDirector에 파일을 생성한다. 이때 HDFS에 저장된 색인 파일의 이름과 동일한 이름으로 생성한다.

**IndexOutput**indxout = rdir.**createOutput**(filelst[i]);

    // 바이트 배열로부터 RAMDirectory의 파일로 데이터를 쓴다.

    indxout.writeBytes(bytarr,bytarr.length);  
    indxout.flush();          
    indxout.close();                  
}

filereader.close();

이제 필요한 색인 파일은 모두 RAMDirectory(또는 메모리)에 존재하게 된다. 따라서 해당 색인 파일에 대해 검색을 직접 실행할 수 있다. RAMDirectory에 대한 검색 코드는 로컬 파일 시스템에 대해 검색했던 코드와 유사하다. 유일한 차이점은 **Searcher** 객체를 생성할 때, 로컬 파일 시스템의 디렉토리 경로가 아니라 RAMDirectory 객체(rdir)를 사용해서 생성한다는 점이다.

Searcher searcher = new **IndexSearcher**(**rdir**);  
Analyzer analyzer = new StandardAnalyzer();  
  
System.out.println("Total Documents = "+searcher.maxDoc()) ;  
QueryParser parser = new QueryParser("time", analyzer);  
  
Query query = parser.parse("02\\:24\\:04");  
  
Hits hits = searcher.search(query);  
  
System.out.println("Number of matching documents = "+ hits.length());  
  
for (int i = 0; i < hits.length(); i++) {  
    Document doc = hits.doc(i);  
    System.out.println(doc.get("date")+" "+ doc.get("time")+ " "+  
    doc.get("cs-method")+ " "+ doc.get("cs-uri")+ " "+ doc.get("sc-status")+ " "+ doc.get("time-taken"));  
}

아래의 출력결과에서 보다시피, "time" 필드에 대해 "02:\\:24\\:04." 텍스트를 사용해서 검색을 수행한다. 따라서 코드가 실행되면, "time"에 "02:\\:24\\:04."가 포함된 문서(또는 행)이 결과로 출력된다.

Total Documents = 11  
Number of matching documents = 4  
2010-04-21 02:24:04 GET /blank 304 233  
2010-04-21 02:24:04 GET /blank 500 567  
2010-04-21 02:24:04 GET /blank 200 897  
2010-04-21 02:24:04 POST /blank 200 567

**결론**

HDFS와 같은 분산 파일 시스템은 오늘날 만들어지는  방대한 량의 데이터를 저장하고 처리할 수 있는 막강한 도구다. 메모리 기반으로 색인하고 검색하는 방식을 사용하면, 산더미같은 데이터 중에서 정말로 원하는 데이터를 좀더 쉽게 찾을 수 있게 된다.

## 부록 7. Splunk

== Splunk 설치하기 on 리눅스

**Splunk란**

Splunk Enterprise를 사용하면 어플리케이션, 서버, 장비가 물리적이던 아니면 클라우드 환경의 가상이든 관계없이  빠르게 생성되는 데이터를 모두 수집하고 색인하여 활용할 수 있다. 어플리케이션에 장애가 생기거나 보안 사고를 조사해야 할 때 시간 단위 또는 일단위가 아닌 수분 내로 해결 가능하다. 이를 통해 서비스 처리가 느려지거나 아예 중단되는 사태를 방지할 수 있다. 따라서 적은 비용을 지불하여 컴플라이언스를 준수할 수 있을뿐만 아니라, 사업에 대한 새로운 통찰력을 얻을 수 있다.

Splunk Enterprise 체험판은 다운로드하면 60동안 무료로 사용할 수 있으며, 하루에 500MB의 데이터를 색인할 수 있다. 무료버전에는 제약사항이 존재하며 상세한 내용은 다음과 같다.

[Free vs. Enterprise](http://www.splunk.com/view/free-vs-enterprise/SP-CAAAE8W)

**다운로드**

[다운로드 페이지](http://www.splunk.com/download?r=header)에서 Splunk를 다운로드 한다.

여기에서는 splunk-5.0.1-143156-Linux-x86\_64.tgz 버전을 다운로드했다.

**설치하기**

적당한 위치에서 압축을 푼다.

$ tar xvzf splunk-5.0.1-143156-Linux-x86\_64.tgz

아래의 명령어를 실행하여 Splunk를 기동한다.

$ $SPLUNK\_HOME/bin/splunk start

중지하려면 아래와 같이 stop 명령을 사용한다.

$ $SPLUNK\_HOME/bin/splunk start

약관이 나오며, 맨 마지막에 동의한다(y)

기동이 완료되면 아래와 같은 메시지가 나온다.

The Splunk web interface is at **http://[hostname]:8000**

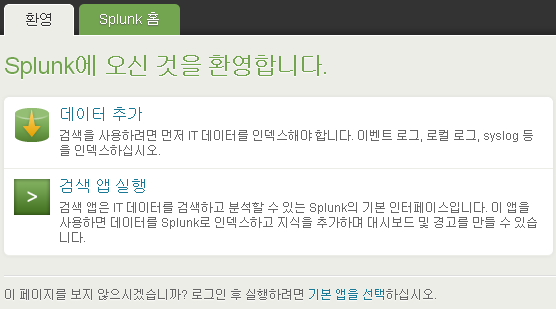
**웹 콘솔 접속하기**

웹 콘솔에 처음 접속하면 아래와 같은 화면이 나온다.



처음 접속하는 경우 아이디와 패스워드는  admin/changeme이며, 로그인을 한 후 패스워드를 변경한다.

로그인이 되면 아래와 같은 화면이 나온다.

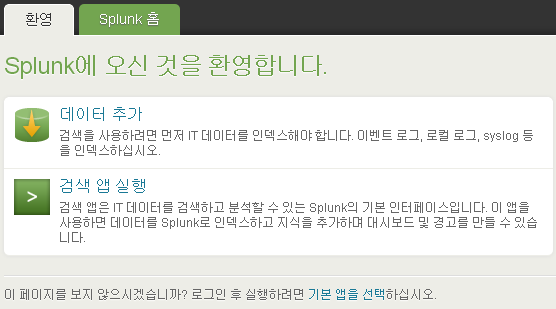


**참고자료**

* [Download and install Splunk](http://docs.splunk.com/Documentation/Splunk/latest/Tutorial/InstallSplunk)
* [Install on Linux](http://docs.splunk.com/Documentation/Splunk/5.0.1/Installation/InstallonLinux)

== Splunk에 데이터 추가하기 - 로컬 파일

**[데이터 추가]**를 선택한다.



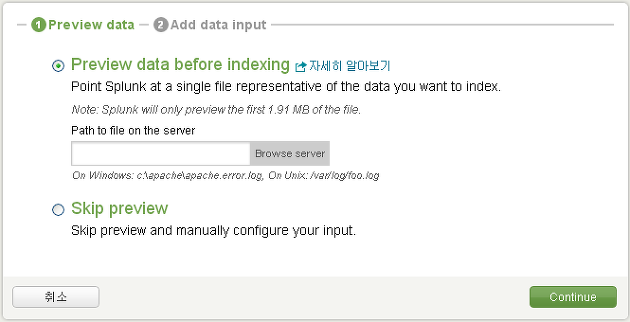
**[파일 또는 파일의 디렉터리]**를 선택한다.



**[이  Splunk 서버에서 파일 사용]**을 선택한다.

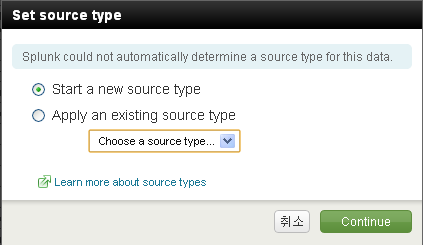


추가할 파일을 선택한다.

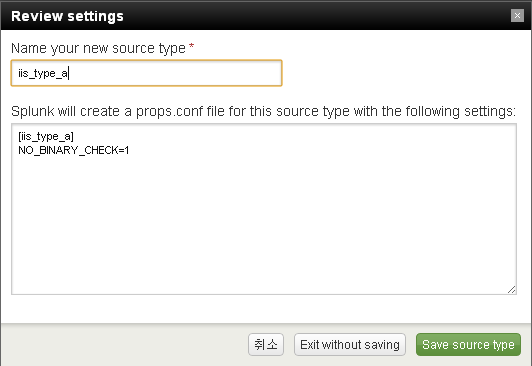


source type을 선택한다.

여기에서는 **[Start a new source type]**을 선택한다.



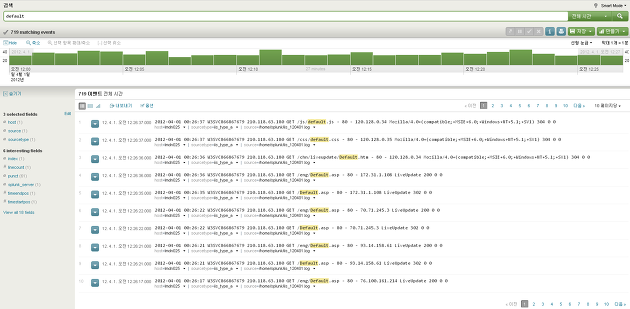
source type 이름을 지정한 후 저장한다.



추가한 데이터가 인덱스된다.



검색을 하면 아래와 같이 검색된 결과가 나온다.

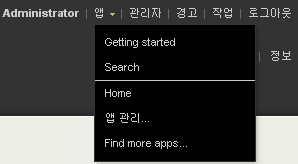


== Splunk와 하둡 연동하기 – HadoopConnect

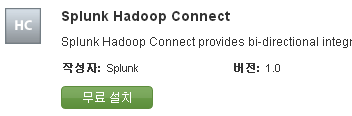
Splunk는 플러그앤플레이 형태로 필요한 모듈을 설치할 수 있으며, 이러한 모듈을 앱(App)이라고 부른다. 이번에는 Splunk와 하둡을 연동하기 위한 HadoopConnect 앱을 설치하고, 하둡에 저장된 파일을 검색하는 기능을 사용해보려고 한다.

**HadoopConnect 앱 설치하기**

[앱] 메뉴에서 [Find more apps...] 메뉴를 선택한다.

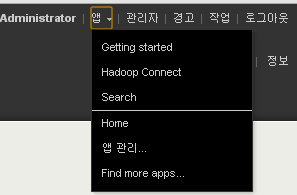


Hadoop으로 검색한 후, [Splunk Hadoop Connect]를 설치한다.

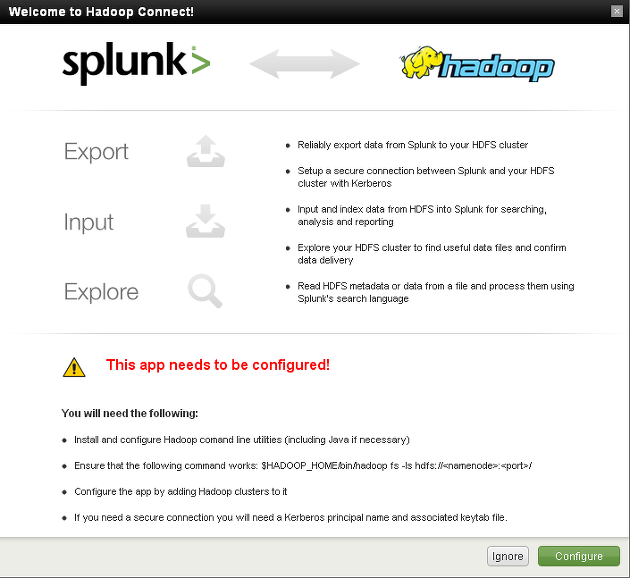


설치가 완료되면 Splunk를 재시작한 후 다시 로그인한다.

[앱] 메뉴에 지금 설치한 Hadoop Connect 메뉴가 생기며, 해당 메뉴를 선택한다.



HadoopConnect를 사용하기 위해서는 하둡 클러스터와 관련된 정보를 설정해야 한다. [Configure]를 선택한다.



아직 등록된 하둡 클러스터가 없으므로, [Add Cluster]를 선택한다.



기본적으로 4가지 정보를 제공해야 한다.



* **HDFS URI**  
  하둡 네임노드 URI. 하둡 클러스터의 core-site.xml 파일의 fs.default.name에 해당하는 값이다.
* **HADOOP\_HOME**  
  하둡 네임노드가 설치된 홈 디렉토리다.
* **JAVA\_HOME**  
  자바가 설치된 홈 디렉토리다. hadoop-env.sh 파일의 JAVA\_HOME에 해당하는 값이다.
* **Namenode HTTP Port**  
  네임노드의 HTTP 포트다. 기본적으로 50070을 사용한다.
* **Set Security**  
  커버로스 인증을 사용하는 경우에 설정한다(여기에서는 설정하지 않았다)

이제 추가된 하둡 클러스터를 확인할 수 있다.



**하둡 파일 검색하기**

[Explore]를 선택하면 아래와 같이 하둡 네임노드 콘솔에서 보는 것과 유사한 화면이 나온다. 디렉토리를 탐색할 수 있으며, 파일인 경우에는 아래와 같이 [Search] 메뉴가 표시된다.

http://cfile28.uf.tistory.com/image/154E8A4350F5026C21376B

[Search] 메뉴를 선택하면 아래와 같이 검색된 결과가 나온다.

자동으로 생성된 검색 명령어는 다음과 같다.

| **hdfs read**hdfs://lmms001:9000/user/hive/warehouse/beaver.db/tbl\_web\_w3c\_0001/20121019

즉 hdfs의 해당 파일을 read하는 명령어를 실행한다.

Splunk의 경우 인덱스하는 시점이 아닌 검색하는 시점에 필드를 추출할 수 있다. 예를 들어 탭을 구분자로 사용하여, 분리된는 각 필드의 이름을 아래와 같이 지정할 수 있다.

| hdfs read hdfs://lmms001:9000/user/hive/warehouse/beaver.db/tbl\_web\_w3c\_0001/20121019 **delim="\t" fields**="cid,aip,timestamp,date,time,cip,method,url,port"

뿐만 아니라 파이프라인을 사용하여 정렬하거나 집계 명령문을 실행할 수도 있다.

| hdfs read hdfs://lmms001:9000/user/hive/warehouse/beaver.db/tbl\_web\_w3c\_0001/20121019 delim="\t" fields="cid,aip,timestamp,date,time,cip,method,url,port" **| sort url**

| hdfs read hdfs://lmms001:9000/user/hive/warehouse/beaver.db/tbl\_web\_w3c\_0001/20121019 delim="\t" fields="cid,aip,timestamp,date,time,cip,method,url,port" | sort url **| chart sum**(port) as sum\_cip **by** cip

**Splunk 앱 삭제하기**

Splunk 웹 콘솔에서는 앱을 활성화/비활성화할 수는 있지만 앱을 삭제할 수는 없다. 삭제하려면 Spunk가 설치된 장비에 접속하여 설치된 앱의 디렉토리를 삭제해야 한다.

예를 들어 지금 설치한 HadoopConnect를 삭제하려면 아래와 같이 2개의 디렉터리에서 앱을 삭제해야 한다.

$ rm -rf $SPLUNK\_HOME/etc/users/admin/HadoopConnect/

$ rm -rf $SPLUNK\_HOME/etc/apps/HadoopConnect/

**참고자료**

* [Connecting Splunk and Hadoop](http://blogs.splunk.com/2012/12/20/connecting-splunk-and-hadoop/)

## 부록 8. 플럼 OG vs. 플럼 NG

플럼 NG는 플럼 NG 리팩토링 이슈에 따라 기존의 플럼, 즉 플럼 OG를 리팩토링한 프로젝트다. 플럼 OG에는 많은 제약사항이 있었다. 이 같은 플럼 OG의 한계를 개선하기 위해 플럼 NG가 만들어졌으며, 중앙 집중식에서 분산형으로 아키텍처를 완전히 달리 가져갔다. 플럼 OG에서는 에이전트와 컬렉터를 관리하는 마스터가 있으며, 마스터 노드를 통해 각 노드를 원격에서 중앙집중식으로 관리할 수 있다. 반면 플럼 NG에서는 아키텍처를 단순화하기 위해 마스터 노드를 없앴다. 각 노드는 로컬에 위치한 설정 파일에서만 가능하며, 따라서 원격 설정이 불가능하다.

플럼 NG와 플럼 OG 사이의 주요 차이점은 다음과 같다.

* 노드를 소스와 싱크로 구분한다는 점에는 변함이 없으나, NG에서는 채널(channel)을 통해 소스와 싱크가 서로 연결되어 있다.
* 채널은 플러그인 형태로 구성되며, 채널의 지속성을 상황에 맞게 설정할 수 있다. 만약 지속성은 없지만 빠르게 처리하고 싶다면 인메모리 채널을 사용할 수 있다. 반면 이벤트가 지속성을 가져야 한다면 파일 기반의 채널을 사용할 수 있다.
* NG 버전에서는 논리 노드와 물리 노드의 구분이 없다. 물리 노드는 단순하 에이전트라고 부르며, 각 에이전트에서 0개 이상의 소스와 싱크를 실행할 수 있다.
* NG 버전에서는 마스터 노드가 없으며, 따라서 주키퍼에 대한 의존성을 없앴다. 대신 설정 파일을 사용해서 간단히 설정할 수 있다.
* NG 버전에서는 모든 요소가 플러그인이다. 즉 채널, 소스, 싱크, 인터셉터, 싱크 프로세서 등 모든 요소가 플러그인 가능한 구성요소다.

## 부록 9. 샘플 웹로그

<http://ita.ee.lbl.gov/html/contrib/Sask-HTTP.html>

에서 다운로드

Saskatchewan-HTTP

**Description**

This trace contains seven month's worth of all HTTP requests to the University of Saskatchewan's WWW server. The University of Saskatchewan is located in Saskatoon, Saskatchewan, Canada.

**Format**

The logs are an ASCII file with one line per request, with the following columns:

1. **host** making the request. A hostname when possible, otherwise the Internet address if the name could not be looked up.
2. **timestamp** in the format "DAY MON DD HH:MM:SS YYYY", where **DAY** is the day of the week, **MON** is the name of the month, **DD** is the day of the month, **HH:MM:SS** is the time of day using a 24-hour clock, and **YYYY** is the year. The timezone is -0400.
3. **request** given in quotes.
4. **HTTP reply code**.
5. **bytes in the reply**.

**Measurement**

The first log was collected from 00:00:00 June 1, 1995 through 23:59:59 December 31, 1995, a total of 214 days. In this seven month period there were 2,408,625 requests. Timestamps have 1 second resolution.

**Privacy**

The logs fully preserve the originating host and HTTP request. Please do not however attempt any analysis beyond general traffic patterns.

**Acknowledgements**

The logs was collected by Earl Fogel of the University of Saskatchewan,, and contributed by Martin Arlitt (mfa126@cs.usask.ca) and Carey Williamson (carey@cs.usask.ca) of the University of Saskatchewan.

**Publications**

This is one of six data sets analyzed in an upcoming paper by   
M. Arlitt and C. Williamson, entitled ``Web Server Workload Characterization: The Search for Invariants'', to appear in the proceedings of the 1996 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, Philadelphia, PA, May 23-26, 1996. An [extended version](ftp://ftp.cs.usask.ca/pub/discus/paper.96-3.ps.Z) of this paper is available on-line; see also the [DISCUS home page](http://www.cs.usask.ca/projects/discus/discus.html) and the group's [publications](http://www.cs.usask.ca/projects/discus/discus_pub.html).

**Related**

Permission has been granted to make four of the six data sets discussed in ``Web Server Workload Characterization: The Search for Invariants'' available. The four data sets are: [Calgary-HTTP](http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html) , [ClarkNet-HTTP](http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html) , [NASA-HTTP](http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html) , and [Saskatchewan-HTTP](http://ita.ee.lbl.gov/html/contrib/Sask-HTTP.html) .

**Restrictions**

The traces may be freely redistributed.

**Distribution**

Available from the Archive in [**Access log, ASCII format, 30.5 MB gzip compressed**](ftp://ita.ee.lbl.gov/traces/usask_access_log.gz), 233.4 MB uncompressed, and [**Abort log, ASCII format, 370 KB gzip compressed**](ftp://ita.ee.lbl.gov/traces/usask_abort_log.gz), 2.7 MB uncompressed.

## 부록 10. 테스트 주도로 하둡 맵리듀스 프로그래밍하기

**1. 개요**  
이 글에서는 테스트 주도로 하둡 맵리듀스 프로그램을 개발하는 방법을 설명합니다. 개발할 문제 영역은 하둡 맵리듀스 튜토리얼에서 다루고 있는 "Word Count" 예제입니다.   
  
**1.1 준비사항**

* **하둡 설치**  
  로컬 머신에서 맵리듀스 프로그램을 개발하고 테스트하는 방법을 설명하므로, 로컬 머신에 하둡이 설치되어 있어야 합니다. 하둡 클러스터에서 테스트하기 전에 먼저 로컬에서 테스트하는 방법이므로, 로컬 머신에는 하둡이 독립 모드 또는 의사 분산 모드로 설치되어 있어야 합니다. 하둡을 의사 분산 모드로 설치하는 방법을 보시려면 [여기](http://socurites.com/68)를 참고하시기 바랍니다.
* **jUnit 4와 모키토(Mockito)**  
  또한 테스팅 프레임워크로는 jUnit 4와 모키토 1.8.5를 사용합니다. 모키로 사용법을 모르시는 분은 [여기](http://socurites.com/24)를 참고하시기 바랍니다.
* **메이븐(Maven)**  
  그리고 프로젝트는 메이븐을 사용해 생성합니다. 다시 말해 메이븐을 사용해서 의존성이 있는 라이브러리를 관리하고, 프로젝트를 구성합니다. 메이븐에 대해서는 [아파치 메이븐 사이트](http://maven.apache.org/)를 참고하세요. 여기에서는 메이븐의 특수한 기능을 사용하지 않으므로 알지 못하더라도 진행하는데 큰 어려움이 없을리라고 봅니다.
* **이클립스와 m2eclipse 플러그인**  
  마지막으로, 이클립스를 IDE로 사용하며 여기에서는 인디고(Indigo) 버전을 사용합니다. 그리고 메이븐 플러그인을 설치해두어야 합니다. 여기에서는 m2eclipse(http://download.eclipse.org/technology/m2e/releases) 플러그인을 설치했습니다.

**2. 메이븐 프로젝트 구성하기**  
**2.1 메이븐 프로젝트 생성하기**  
먼저 메이븐 프로젝트를 생성합니다. 이클립스에서 메이븐 프로젝트를 생성하려면 반드시 메이븐 플러그인이 먼저 설치되어 있어야 합니다. 아래와 같이 메이븐 프로젝트를 생성할 수 있습니다. [Maven Project]를 선택한 후, [Next]를 클릭합니다.  
  
  
"New Maven Project' 창에서 기본 설정 그대로 둔 채, [Next] 버튼을 클릭합니다.  
  
  
메이븐 프로젝트 아케타입(Archetype) 중에서, "maven-archetype-quickstart"를 선택한 후, [Next]를 클릭합니다.  
  
  
"Group Id"와 "Artifcat Id"를 입력합니다. 여기에서는 Group Id로 "com.socurites"를 Artifact Id로는 "example.hadoop.wordcount"를 입력햇습니다. 이렇게 입력하면 자동으로 패키지(Package) 명이 구성됩니다. 패키지 명은 [Group Id].[Artifcat Id]로 구성됩니다. 또한 Artifcat Id는 이클립스에서 생성될 프로젝트 명이 됩니다. 이제 [Finish]를 클릭합니다.  
  
  
이 작업이 끝나면 이클립스에 워크스페이스에 "example.hadoop.wordcount"라는 이름의 프로젝트가 생성됩니다. 여기에서 자동으로 생성된 App.java와 AppTest.java는 필요 없으므로 삭제합니다.  
  
 **2.2 의존성 설정하기**  
이제 프로젝트의 라이브러리 의존성을 설정합니다. 여기에서 사용할 라이브러리는 총 3개로 다음과 같습니다.

* **jUnit 4.1**  
  단위 테스트를 만들 때 사용할 라이브러리로, 테스트 목적으로만 사용합니다.
* **mockito 1.8.5**  
  Mock 객체를 만들 때 사용할 라이브러리로, 테스트 목적으로만 사용합니다.
* **Hadoop 0.20.2**  
  MapReduce 프로그램을 개발할 때 사용할 라이브러리로, 컴파일할 목적으로만 사용합니다.

이제 pom.xml 파일을 열어, [Dependencies] 탭으로 이동해서 의존성을 설정합니다.  
먼저 jUnit 4.1을 추가해 봅시다. 기본적으로 jUnit이 3.8.1이 등록되어 있으므로 [Remove] 버튼을 클릭해서 삭제합니다.   
이제 [Add...] 버튼을 클릭합니다. "junit"을 검색어로 입력하면 많은 수의 라이브러리가 나오는데 여기에서 아래 그림과 같이 junit의 "4.10 [jar]"을 선택합니다. 그리고 jUnit은 테스트 단계에서만 사용할 것이므로 Scope를 "test"로 설정합니다. [OK] 버튼을 클릭합니다.  
  
  
마찬가지로 mockito 1.8.5와 hadoop-core 0.20.2를 추가합니다. 이때 hadoop-core는 Scope를 "compile"로 설정해서 추가해야 합니다.  
  
  
이제 맵리듀스 프로그램을 개발할 준비가 끝났습니다.   
  
  
**3. Mapper 개발하기**  
**3.1 Mapper 테스트 케이스 만들기**  
먼저 맵 함수에 대한 테스트 케이스를 생성하고, 실패하도록 만들어야 합니다.   
"src/test/java"의 com.socurites.example.hadoop.wordcount.mapper 패키지에 WordCountMapperTest라는 이름으로 매퍼에 대한 테스크 케이스를 생성합니다.   
  
  
  
WordCountMapperTest는 아래와 같이 작성합니다.

package com.socurites.example.hadoop.wordcount.mapper;  
  
import static org.mockito.Mockito.mock;  
import static org.mockito.Mockito.times;  
import static org.mockito.Mockito.verify;  
import java.io.IOException;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper.Context;  
import org.junit.Test;  
import com.socurites.example.hadoop.wordcount.mapper.WordCountMapper;  
  
public class **WordCountMapperTest** {  
    @Test  
    public void **processValidRecord**() throws IOException, InterruptedException {  
        **WordCountMapper** mapper = new WordCountMapper();  
          
        LongWritable key = null;  
        Text value = new Text("*2005년 12월 8일 야후! 코리아의 제안으로 Yahoo! Answers 서비스 개시*");  
        Context context = **mock(Context.class)**;  
          
        **mapper.map(key, value, context);**  
          
        **verify(context, times(1)).write(new Text("2005년"), new IntWritable(1));**  
    }  
}

코드에서 보는 바와 같이 Context 클래스에 대해 Mock 객체를 생성한 후, map 함수를 호출합니다. 그리고, context 객체에서 wirte 함수가 1번 호출되고, 이때 "2005년"을 키로 가지는 값이 1이었는지 검증합니다.  
  
이렇게 테스트케이스를 작성하고 나면 컴파일이 되지 않습니다. 이제 테스트 케이스가 컴파일 되도록 맏들어야 합니다.  
  
**3.2 Mapper 테스트 컴파일되도록 만들기**  
이제 Mapper 테스트가 컴파일 되도록 만들어야 합니다. WordCountMapper 클래스를 "src/main/java"의 "com.socurites.example.hadoop.wordcount.mapper" 패키지에 생성한 후, map 메서드를 구현합니다. 지금 당장은 아래와 같이 비어있는 상태로 만들어 둡니다.

package com.socurites.example.hadoop.wordcount.mapper;  
  
import java.io.IOException;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;  
  
public class **WordCountMapper** extends  
        Mapper<LongWritable, Text, Text, IntWritable> {  
  
    @Override  
    public void **map**(LongWritable key, Text value,  
            Context context)  
            throws IOException, InterruptedException {  
    }  
}

테스트케이스를 실행해보면 아래와 같이 빨간불이 뜹니다. 즉 테스트가 실패합니다.  
  
  
보다시피 context.write(2005년, 1)이 한번 호출되어야 하나, 호출되지 않았음을 확인할 수 있습니다. 이제 테스트를 성공하도록 만들어야 합니다.  
  
**3.3. Mapper 테스트 녹색불 뜨도록 만들기**  
WordCountMapper 클래스의 map 함수를 아래와 같이 구현합니다.

package com.socurites.example.hadoop.wordcount.mapper;  
  
import java.io.IOException;  
import java.util.StringTokenizer;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;  
  
public class WordCountMapper extends  
        Mapper<LongWritable, Text, Text, IntWritable> {  
**private final IntWritable one = new IntWritable(1);**  
  
    @Override  
    public void **map**(LongWritable key, Text value,  
            Context context)  
            throws IOException, InterruptedException {  
**StringTokenizer tokenizer = new StringTokenizer(value.toString());  
          
        while ( tokenizer.hasMoreTokens() ) {  
            context.write(new Text(tokenizer.nextToken()), one);  
        }**  
    }  
}

이렇게 구현한 후, 테스트를 실행하면 녹색불이 뜨며 테스트가 성공합니다. 이렇게 해서 Mapper 구현이 끝납니다.  
  
 **4. Reducer 개발하기**  
**4.1 Reducer 테스트 케이스 만들기**  
이제 리듀스 함수에 대한 테스트 케이스를 생성하고, 실패하도록 만들어야 합니다.   
"src/test/java"의 com.socurites.example.hadoop.wordcount.reducer 패키지에 WordCountReducerTest라는 이름으로 리듀서에 대한 테스크 케이스를 생성합니다.

package com.socurites.example.hadoop.wordcount.reducer;  
  
import static org.mockito.Mockito.mock;  
import static org.mockito.Mockito.verify;  
import java.io.IOException;  
import java.util.Arrays;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer.Context;  
import org.junit.Test;  
import com.socurites.example.hadoop.wordcount.reducer.WordCountReducer;  
  
public class WordCountReducerTest {  
    @Test  
    public void **processValidRecord**() throws IOException, InterruptedException {  
        **WordCountReducer** reducer = new WordCountReducer();  
          
        Text key = new Text("2005년");  
        Iterable<IntWritable> values = Arrays.asList(new IntWritable(1), new IntWritable(1));  
        Context context = **mock(Context.class);**  
          
        **reducer.reduce(key, values, context);**  
          
        **verify(context).write(key, new IntWritable(2));**  
    }  
}

보다시피 테스트에서는 "2005"년을 키로 하여, 리듀스한 값이 2가 되도록 구성합니다. 매퍼 테스트와 마찬가지로 테스트케이스를 작성하고 나면 컴파일이 되지 않습니다. 이제 테스트 케이스가 컴파일 되도록 맏들어야 합니다.  
  
**4.2 Reducer 테스트 컴파일 되도록 만들기**  
 WordCountReducer 클래스를 "src/main/java"의 "com.socurites.example.hadoop.wordcount.reducer" 패키지에 생성한 후, reduce 메서드를 구현합니다. 지금 당장은 아래와 같이 비어있는 상태로 만들어 둡니다.

package com.socurites.example.hadoop.wordcount.reducer;  
  
import java.io.IOException;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;  
  
public class **WordCountReducer** extends Reducer<Text, IntWritable, Text, IntWritable> {  
    @Override  
    protected void **reduce**(Text key, Iterable<IntWritable> values,  
            Context context)  
            throws IOException, InterruptedException {  
    }      
}

테스트케이스를 실행해보면 아래와 같이 빨간불이 뜹니다. 즉 테스트가 실패합니다.  
  
  
보는 바와 같이 "2005년"을 키로 값이 2가 되도록 context.write()가 호출되어야 하지만 호출되지 않았음을 알 수 있습니다.  
 **4.3 Reducer 테스트 녹색불 뜨도록 만들기**  
이제 Reducer 테스트가 성공하도록 reduce 메서드를 아래와 같이 구현합니다.

package com.socurites.example.hadoop.wordcount.reducer;  
  
import java.io.IOException;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;  
  
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {  
    @Override  
    protected void **reduce**(Text key, Iterable<IntWritable> values,  
            Context context)  
            throws IOException, InterruptedException {  
**int sum = 0;  
        for ( IntWritable value : values ) {  
            sum = sum + value.get();  
        }  
          
        context.write(key, new IntWritable(sum));**  
    }  
}

이렇게 구현한 후, 테스트를 실행하면 녹색불이 뜨며 테스트가 성공합니다. 이렇게 해서 Reducer 구현이 끝납니다.  
  
이처럼 맵과 리듀스 함수에 대한 단위 테스트가 끝나면 하둡 클러스터에 맵리듀스 프로그램을 배포할 수 있는 수준이 됩니다. 하지만 그전에 단순한 테스트 값이 아니라, 실제 테스트 데이터 중 작은 량을 대상으로 로컬 머신에서 맵리듀스 프로그램을 테스트해 봐야 합니다.  
  
  
**5. Driver 만들기**  
맵리듀스 잡을 실제로 수행할 Driver 클래스를 만들어야 합니다. 로컬 머신에서 테스트할 수 있도록 Configured를 상속받고, Tool 인터페이스를 구현합니다. WordCountDriver 클래스를 "src/main/java"의 "com.socurites.example.hadoop.wordcount.driver" 패키지에 생성한 후, 아래와 같이 구현합니다.

package com.socurites.example.hadoop.wordcount.driver;  
  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;  
import com.socurites.example.hadoop.wordcount.mapper.WordCountMapper;  
import com.socurites.example.hadoop.wordcount.reducer.WordCountReducer;  
  
public class WordCountDriver**extends Configured implements Tool** {  
    public int **run**(String[] args) throws Exception {  
        ***// Job생성하기***  
        Job job = new Job(this.getConf(), "Word Count");  
        job.setJarByClass(this.getClass());  
                  
       ***// Mapper 설정하기***  
        job.setMapperClass(WordCountMapper.class);  
       ***// Reducer 설정하기***  
        job.setReducerClass(WordCountReducer.class);  
          
        ***// 입력 경로 설정하기***  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
       ***// 출력 경로 설정하기***  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
          
       ***// 입력 포맷 설정하기***  
        job.setInputFormatClass(TextInputFormat.class);  
          
       ***// 출력 포맷 설정하기***  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
          
        job.waitForCompletion(true);  
          
        return 0;  
    }  
      
    public static void **main**(String[] args) throws Exception {  
        int exitCode = 0;  
        if ( !hasEnoughArguments(args) ) {  
            System.err.println("사용법: WordCountDriver [generic options] <input> <output>");  
            ToolRunner.printGenericCommandUsage(System.err);  
            exitCode = -1;  
        } else {  
            exitCode = **ToolRunner.run**(new WordCountDriver(), args);  
        }  
          
        System.exit(exitCode);  
    }  
      
    private static boolean hasEnoughArguments(String[] consoleArgs) {  
        if ( consoleArgs.length == 2 ) {  
            return true;  
        }  
          
        return false;  
    }  
}

**6. 실행하기**  
**6.1 입력 데이터 준비하기**  
먼저 입력 데이터를 준비합니다. 실제로 사용할 데이터에서 일부부분을 파일로 만듭니다. 여기에서는 "src/main/resources"에 "testdata.txt"라는 이름으로 파일을 생성했습니다.  
  
  
  
테스트에서 사용한 testdata.txt 파일의 내용은 다음과 같습니다.

    1997년 9월 1일 야후! 코리아 서비스 개시, 염진섭 사장 취임  
    1998년 9월 서비스 개시 1년 만에 1일 300만 페이지뷰 돌파  
    1999년 5월 미국 ABC 가입으로 트래픽 공인 인증  
    1999년 9월 국내 최초 하루 2000만 페이지뷰 돌파  
    2001년 9월 2002 FIFA 월드컵 TM 공식 후원사 선정  
    2004년 2월 자체 검색 엔진 YST(Yahoo! Search Technology) 적용  
    2004년 7월 지역 검색 '거기' 서비스 개시  
    2005년 10월 멀티미디어 검색 '야미' 서비스 개시  
    2005년 12월 2006년 독일 월드컵 공식 홈페이지 서비스 개시  
    2005년 12월 8일 야후! 코리아의 제안으로 Yahoo! Answers 서비스 개시  
    2006년 6월 G마켓 지분 10% 인수  
    2007년 4월 김정우 총괄대표 취임  
    2009년 1월 김대선 총괄대표 취임

**6.2 로컬 머신을 위한 설정파일 만들기**  
로컬 머신에서 테스트하는 경우 하둡 파일 시스템이 아니라 일반적인 파일 시스템을 사용하고자 한다. 그리고 맵리듀스 프로그램도 로멀 머신에서 실행하고자 한다. 이를 위한 설정 파일을 "hadoop-local.xml"이라는 이름으로 만든다. 설정 내용은 다음과 같다.

<?xml version="1.0"?>  
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>  
  
<configuration>  
  <property>  
    <name>**fs.default.name**</name>  
    <value>**file:///**</value>  
  </property>  
  
  <property>  
    <name>**mapred.job.tracker**</name>  
    <value>**local**</value>  
  </property>  
</configuration>

**6.3 로컬 머신에서 실행하기**  
이제 쉘 프롬프트로 이동해서, 이클립스 워크스페이스로 이동한 후, 컴파일된 클래스 파일이 있는 최상위 디렉토리로 이동합니다.

cd [eclipse workspace]/example.hadoop.workcount/target/classes

하둡의 CLASS\_PATH를 현재 디렉토리로 설정합니다.

export HADOOP\_CLASSPATH=.

아래와 같이 하둡 명령어를 실행합니다.

hadoop **com.socurites.example.hadoop.wordcount.driver.WordCountDriver** -conf [hadoop-local.xml 파일이 있는 디렉토리]/hadoop-local.xml ./testdata.txt output

이 명령어를 실행하면 맵 리듀스 프로그램을 외부 잡트래커를 액세스 하지 않고 내부 프로세스로 실행한다. 그리고 ./testdata.txt 파일을 파일시스템에서 입력으로 읽어서, 그 결과를 output 디렉토리에 저장합니다. 이때 output 디렉토리가 있는 경우 데이터를 덮어쓰는 경우를 방지하기 위해 예외가 발생하므로, 존재하지 않는 디렉토리를 출력 디렉토리로 지정해야 합니다.  
  
실제로 디렉토리를 보면 output 디렉토리가 생겻으며, 그 안에 part-r-0000 파일이 생겼음을 확인할 수 있습니다. 그리고 part-r-0000 파일을 열어보면 맵리듀스 프로그램의 결과가 제대로 생성되었음을 확인할 수 있습니다.  
  
  
**[부록 A: 소스 프로젝트]**  
개발한 소스 프로젝트를 아래에서 다운로드 받으신 후, 이클립스에서 임포트하시기 바랍니다.

http://i1.daumcdn.net/icon/editor/p_zip_s.gif wordcount.zip

이때 이클립스에 메이븐 플러그인이 설치되어 있어야만 의존성이 있는 라이브러리를 업데이트 받을 수 있습니다.  
  
**[참고자료]**

* 클라우드 컴퓨팅 구현 기술 / 에이콘 / 김형준, 조준호, 안성화, 김병준 지음
* 하둡 완벽 가이드 / 한빛 미디어 / 심탁길, 김우현 옮김
* [MapReduce Tutorial](http://hadoop.apache.org/common/docs/current/mapred_tutorial.html)

## 부록 11. 하둡 배포판

하둡 아파치 사이트(<http://hadoop.apache.org>)에서 기본 버전을 다운로드

기본 버전에 있는 버그가 있고, 하둡 에코 시스템간의 상호 호환성에 문제가 있음.

이를 해결하고 보완하기 위해 다양한 하둡 배포판이 만들어짐.

\*\* 시작하세요 하둡 프로그래밍 17페이지 참조

### 클라우데라 CDH

### 호튼웍스 HDP

### MapR 테크놀로지스 MapR Editions

### EMC GreenPlum UAP

### 아마존 EMR

### HStreaming

# 참고자료

**책**

클라우드 컴퓨팅 구현 기술, 김형준 (완)

시작하세요 하둡 프로그래밍

직접 해보는 하둡 프로그래밍

거침없이 배우는 하둡

하둡 완벽 가이드

하이브 완벽 가이드

NoSQL 빅데이터 세상으로 떠나는 간결한 안내서

HBase the Definitive Guide

루신 인 액션

**웹문서**

Flume NG, Getting Started

<https://cwiki.apache.org/confluence/display/FLUME/Getting+Started#GettingStarted-WhatisFlumeNG%3F>

플럼 NG 리팩토링 이슈

<https://issues.apache.org/jira/browse/FLUME-728>

플럼 사용자 가이드

<http://archive.cloudera.com/cdh/3/flume/UserGuide/>