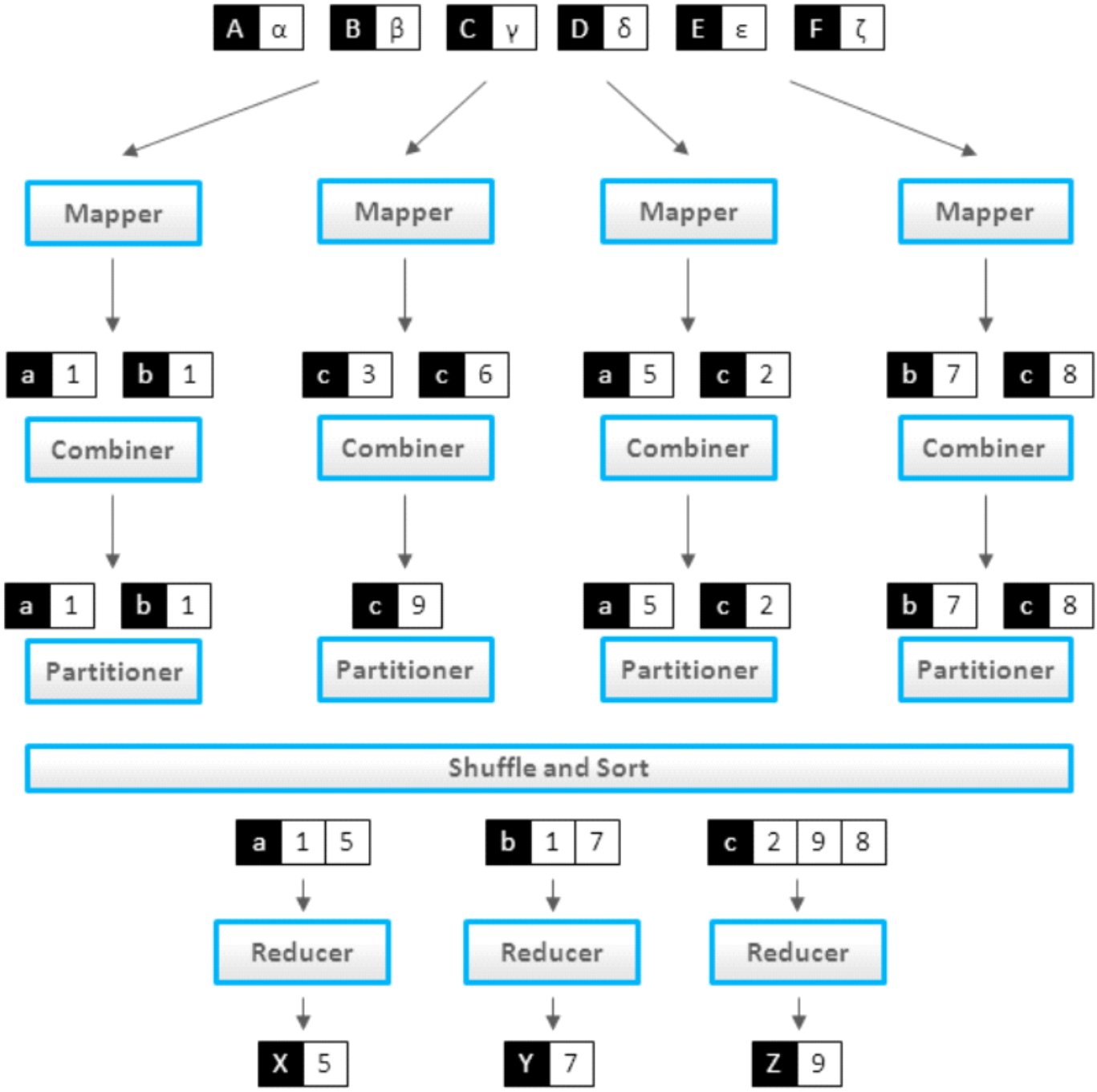
# 개념 익히기

2017년 11월 30일 목요일 오후 12:53




http://bigdataanalyticsnews.com/anatomy-mapreduce-job/

## 개발개발개발

- Home
- Tag
- LocationLog
- Guestbook
- Admin
- Write

2014.06.16 14:36

# Hadoop setPartitionerClass & setGroupingComparatorClass

Hadoop

- **setPartitionerClass & setGroupingComparatorClass**

* **setPartitionerClass**

파티셔너는 맵 태스크의 출력 데이터를 리듀스 태스크의 입력 데이터로 보낼지 결정하고, 이렇게 파티셔닝된 데이터는 맴 태스크의 출력 데이터의 키의 값에 따라 정렬된다. MapReducer에서 사용하는 파티셔너는 반드시 org.apache.hadoop.mapreduce.Partitioner를 상속받아서 구현해야 한다. 이때 파티셔너 설정하는 두 개의 패러미터는 Mapper의 출력 데이터 키와 값에 해당하는 패러미터이다.

예)

```
import org.apache.hadoop.mapreduce.Partitioner;

public class RecPartitioner<KEY, VALUE> extends Partitioner<KEY, VALUE> {

@Override
public int getPartition(KEY key, VALUE value, int numPartitions) {
String strKey = key.toString();

    return (strKey.hashCode() & Integer.MAX_VALUE) % numPartitions;
}

}
```

* **setGroupingComparatorClass**

Groupkey Comparator를 사용해서 같은 Groupkey에 해당하는 모든 데이터를 하나의 Reducer 그룹에서 처리할 수 있다.

예)

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class RecGroupingComparator extends WritableComparator {

protected RecGroupingComparator() {
        super(Text.class, true);
    // TODO Auto-generated constructor stub
}

    public int compare(WritableComparable w1, WritableComparable w2) {
    Text t1 = (Text) w1;
        Text t2 = (Text) w2;

    String t1Key = t1.toString();
    String t2Key = t2.toString();

    return t1key.compareTo(t2key);
```

```
    }
  }
```

<http://devinside.tistory.com/47>에서 삽입

# [참고]Hadoop setPartitionerClass & setGroupingComparatorClass

2017년 11월 30일 목요일 오후 1:09

- **setPartitionerClass & setGroupingComparatorClass**

* **setPartitionerClass**

파티셔너는 맵 태스크의 출력 데이터를 리듀스 태스크의 입력 데이터로 보낼지 결정하고,

이렇게 파티셔닝된 데이터는 맵 태스크의 출력 데이터의 키의 값에 따라 정렬된다.

MapReducer에서 사용하는 파티셔너는 반드시 org.apache.hadoop.mapreduce.Partitioner를

상속받아서 구현해야 한다. 이때 파티셔너 설정하는 두 개의 패러미터는 Mapper의

출력 데이터 키와 값에 해당하는 패러미터이다.

예)

```
import org.apache.hadoop.mapreduce.Partitioner;

public class RecPartitioner<KEY, VALUE> extends Partitioner<KEY, VALUE> {

@Override
public int getPartition(KEY key, VALUE value, int numPartitions) {
String strKey = key.toString();

        return (strKey.hashCode() & Integer.MAX_VALUE) % numPartitions;
}

}
```

* **setGroupingComparatorClass**

Groupkey Comparator를 사용해서 같은 Groupkey에 해당하는 모든 데이터를 하나의 Reducer

그룹에서 처리할 수 있다.

예)

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;
```

```
    public class RecGroupingComparator extends WritableComparator {

protected RecGroupingComparator() {
            super(Text.class, true);
    // TODO Auto-generated constructor stub
}

        public int compare(WritableComparable w1, WritableComparable w2) {
    Text t1 = (Text) w1;
            Text t2 = (Text) w2;

    String t1Key = t1.toString();
    String t2Key = t2.toString();

    return t1key.compareTo(t2key);
        }
    }
```

출처: http://devinside.tistory.com/47 [개발개발개발]

출처: <http://devinside.tistory.com/47>

# [참고]writeable

2017년 11월 30일 목요일 오후 1:16

hdfs에 대해서 개관을 쓸지, pig에 대해서 개관을 쓸지 많은 고민이 있었는데

쌩뚱 맞게 writable에 대해서 쓰게 된다.

프로젝트 개관만 계속 쓰다보니까 약간 좀 그런것도 있고

이번엔 데이터 타입 프로토콜 개관에 대해서 적어보려고 한다.

저번에 보았던 것처럼 hadoop 은 key, value의 pair라고 할 수 있는 데이터에 대하여 작동된다.

많은 예제들은 기본적으로 hadoop 예제를 다룰때 text 를 기반으로 한다.

누군가는 int를 다루고 싶을 수도 있고 누군가는 double을 다루고 싶을 수도 있다.
누군가는 사용자정의의 class를 다루고 싶을 수도 있다.

어떻게 하면 될까?

그것을 정의해주는 class가 Writable 인터페이스이다.

http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Writable.html 를 참조하면

writable은 DataInput과 DataOutput을 기반으로한 시리얼라이즈 오브젝트라고 한다.
간단하기 어떤 key나 어떤 value도 hadoop mr에서 사용하려면 해당 type이 writable으로 구현되어야한다.

writable은 write(DataOutput out)와 readFields(DataInput in)를 정의하여 구현한다.

또한 반드시 compreTo도 정의되어야지만 mr에서 셔플링 될 수 있다는 점을 인지해야한다.

WritableComparable이 Writable에 Comparable을 상속한 인터페이스니까 이것을 사용하여

구현하면 되겠다.

**bulit-in 된 주요 writable들**

이름 : 자바타입
IntWritable : Integer
BooleanWritable : Boolean
ByteWritable : Byte
BytesWritable : byte[]
DoubleWritable : Double
FlaotWritable : Float
LongWritable : Long
MD5Hash : byte[] (md5)
NullWritable : null
Text: String (utf8)

그 외에도 오브젝트Writable이라던가 어레이Writable이라던가 MapWritable등이 있지만 별로 추천하진 않는다.
저런 객체를 다룰때는 Tuple 형으로 하나 새로 만들던가, 사용자정의 class를 사용하는 하는 것을 추천한다.

개인적으로 쓰고 싶은 것만 쓴거니까 (중요도가 따로 있지도 않음) 없는 것에 대해서 이상하게 생각할 필요는 없다.
그외 hbase나 pig, mahout 등 여타 다른 프레임웍도 같이 사용하고 있다면 다른 writable이 있을테지만 적지 않겠다.

**write 함수**

write 함수는 해당 객체를 시리얼라이즈하여 byte 스트림으로 변경 시켜주는 역할을 담당한다.

쉽게 보기 위하여 Text와 IntWritable의 예를 까보자

Text의 경우
public void write(DataOutput out) throws IOException {

```
    WritableUtils.writeVInt(out, length);
    out.write(bytes, 0, length);
}
```

쉽게 해당 text의 길이를 out 스트림에 플래그로 위치시키고 (가변인수로) 그 후 실제 내용을 기록했다.
다시 정리하면 "abc"  라는 문서 내용이 있을때
3abc 라고 시리얼라이즈 시켰다고 보면되겠다. (쉬운 이해를 위해 string의 byte[] 변환은 무시했다)

확장하면 'abc", "sonsworld", "hoho" 이런 형식의 Text 객체들은
3abc9sonsworld4hoho 요런식으로 byte 스트림으로 변환되게 된다

IntWritable의 경우

```
public void write(DataOutput out) throws IOException {
    out.writeInt(value);
}
```

기본 DataOutput이 지원하는 writeInt 함수를 통하여 쉽게 스트림화 했다.

Text와 IntWritable의 차이는 Int의 경우 고정 크기를 차지하는 타입이고 String의 경우는 값마다 크기 차이가 있기 때문에 서로 다른 방식임을 알 수 있다.

**ReadFields 함수**

이번에는 ReadFields(DataInput in) 함수를 살펴보자. 해당  함수는 byte 스트림을 다시 원래 객체로 복원시키는 역할을 하는 함수임을 예측할 수 있다.

쉬운 이해를 위하여 Text와 IntWritable을 다시 까보자.

Text의 경우

```
public void readFields(DataInput in) throws IOException {
    int newLength = WritableUtils.readVInt(in);
    setCapacity(newLength, false);
    in.readFully(bytes, 0, newLength);
    length = newLength;
}
```

우리는 Text가 어떤 형식으로 스트림화 되어있는지 기억한다. 바로 길이, 내용 이다.
역으로 디시리얼라이즈되는 장면을 목격 할 수 있다.

우선 ReadVint 를 하여 길이를 in 에서 추출해낸다. 그후 해당 크기 만큼 추가로 내용을 얻어온 후 맴버에게 값을 할당 시킴으로서 객체가 복원됨을 확인할 수 있다.

더 쉬운 IntWritable을 보도록 하자

```
public void write(DataOutput out) throws IOException {
    out.writeInt(value);
}
```

노 코멘트 ㅋㅋ

우리는 이것으로 맵리듀스(map reduce) 프레임웍 내부에서 데이터 프로토콜을 할 수 있게 되었다.
그러나 위에 적은 것처럼 아직 실제로 사용할 수 는 없는데.. 그건은 아직 해당 객체를 비교할 수 있는 방법이 없기 때문이다.

**compareTo 함수**

해당 함수는 Map 작업이 끝난 후 Key에 맞춰서 데이터를 파티셔닝하고 정렬하고 그룹핑할 때 필요한 함수이다.
이 함수는자바 기본 타입이기 때문에 구미에 맞게 대소비교를 하여주면 된다.

우리는 이것으로 온전히 writable 에 대하여 이해하였다.
pig 와 같이 상당히 고도로 추상화된 프레임웍을 사용한다면 writable에 대한 이해가 없어도 mr을 사용하는데 큰 지장이 없으나

java 환경에서 mr job을 구현하고 있다면 알아두면 매우 유익하다고 생각된다.
개인적으로 Map을 추상화하여 만든 Map<String, Writable> 형식의 Writable을 구현하여 편리하게 사용하고 있다.

기본 MapWritable은 키도 Writable 형식으로 사용해서 좀 비추 ㅎㅎ;

마지막으로
내가 키를 String으로 Value를 Integer로 하고 싶다. 하면
Key = Text

Value = IntWritable로 하면 된다.

나중에 hadoop에서의 job 설정 방법에 대해서 알아볼때 좀 더 다뤄질 것이다.

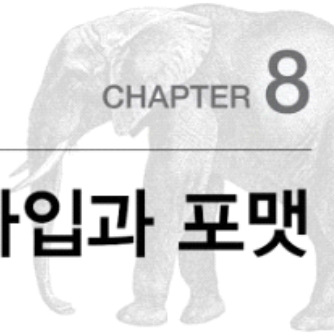
참조
http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Writable.html
http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/WritableComparable.html

출처: http://sonsworld.tistory.com/7 [SonsWorld!]

출처: <http://sonsworld.tistory.com/7>

# [참고]맵리듀스 타입과 포맷

2017년 11월 30일 목요일 오후 1:44



CHAPTER 8

# 맵리듀스 타입과 포맷

맵리듀스의 데이터 처리 모델은 단순하다. 맵과 리듀스 함수의 입력과 출력은 키-값 쌍으로 되어 있다. 이 장에서는 맵리듀스 모델을 자세히 살펴볼 것이다. 특히 단순한 텍스트부터 구조화된 바이너리 객체까지 다양한 포맷의 데이터를 맵리듀스 모델에서 어떻게 처리하는지 살펴보겠다.

## 8.1 맵리듀스 타입

하둡 맵리듀스의 맵과 리듀스 함수의 형식은 다음과 같다.

```
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

일반적으로 맵의 입력키와 값의 타입(K1과 V1)은 맵의 출력 타입(K2와 V2)과 다르다. 하지만 리듀스의 입력은 맵의 출력과 반드시 같은 타입이어야 하며, 리듀스의 출력 타입(K3과 V3)과는 다를 수 있다. 맵리듀스 자바 API의 일반적인 형태는 다음과 같다.

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
      // ...
  }

  protected void map(KEYIN key, VALUEIN value,
      Context context) throws IOException, InterruptedException {
    // ...
  }
}

public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    // ...
  }
  protected void reduce(KEYIN key, Iterable<VALUEIN> values,
      Context context) throws IOException, InterruptedException {
    // ...
  }
}
```

context 객체는 키-값 쌍을 내보낼 때 사용되며, 출력 타입으로 인자화된다. `write( )` 함수의 원형은 다음과 같다.

```
public void write(KEYOUT key, VALUEOUT value)
    throws IOException, InterruptedException
```

Mapper와 Reducer는 별개의 클래스이므로 각각의 타입 매개변수는 서로 다른 유효 범위를 갖는다. 따라서 동일한 이름의 KEYIN 타입 매개변수라도 Mapper와 Reducer의 실제 타입 인자는 서로 다를 수 있다. 예를 들어 이전 장에서 살펴본 최고 기온 예제에서 Mapper는 LongWritable을 Reducer는 Text를 각각 KEYIN으로 사용했다.

마찬가지로 맵의 출력 타입과 리듀스의 입력 타입은 반드시 일치해야 하지만 자바 컴파일러가 이를 강제할 수는 없다.

타입 매개변수와 추상 타입의 이름은 다를 수 있지만(KEYIN과 K1), 그 형태는 동일해야 한다.

컴바인 함수는 Reducer의 구현체로, 리듀스 함수와 동일한 형태를 가진다. 컴바인 함수의 출력 타입은 중간 키-값 타입(K2와 V2)이며, 컴바인의 출력은 리듀스 함수의 입력으로 전달된다.

```
map: (K1, V1) → list(K2, V2)
combiner: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

컴바인 함수와 리듀스 함수는 같은 경우가 많다. 이 경우 K3는 K2와 같고 V3는 V2와 같다.

파티션 함수는 컴바인의 중간 키-값 타입(K2와 V2)을 처리한 후 파티션 인덱스를 반환한다. 실제로 파티션은 순전히 키에 의해 결정된다(그 값은 무시한다).

```
partition: (K2, V2) → integer
```

자바에서는 다음과 같이 구현한다.

```
public abstract class Partitioner<KEY, VALUE> {
  public abstract int getPartition(KEY key, VALUE value, int numPartitions);
}
```

**이전 API에서 맵리듀스 함수의 원형**

이전 API(부록 D 참조)를 보면 맵리듀스 원형은 거의 비슷하고 이전과 새로운 API의 제약 조건도 정확히 같지만 K1, V1 등 타입 매개변수의 이름을 실제로 지정했다.

```
public interface Mapper<K1, V1, K2, V2> extends JobConfigurable, Closeable {
  void map(K1 key, V1 value,
      OutputCollector<K2, V2> output, Reporter reporter) throws IOException;
}
```

```
public interface Reducer<K2, V2, K3, V3> extends JobConfigurable, Closeable {
  void reduce(K2 key, Iterator<V2> values,
      OutputCollector<K3, V3> output, Reporter reporter) throws IOException;
}

public interface Partitioner<K2, V2> extends JobConfigurable {
  int getPartition(K2 key, V2 value, int numPartitions);
}
```

이론은 이쯤에서 접고, 맵리듀스 잡을 설정하는 데 도움이 되는 방법을 살펴보자. [표 8-1]은 새로운 API의 환경 설정 옵션을 요약한 것이다([표 8-2]는 이전 API). 표는 타입을 결정하는 속성과 설정된 타입의 상호 호환성으로 크게 나뉘어 있다.

입력 타입은 InputFormat을 통해 설정된다. 예를 들어 TextInputFormat은 LongWritable 타

입의 키와 Text 타입의 값을 생성한다. 다른 입력 타입은 Job의 메서드(이전 API의 JobConf)를 명시적으로 호출하여 설정한다. 중간 타입을 명시적으로 설정하지 않으면 자동으로 (최종) 출력 타입과 같게 되어 있다. 출력의 기본 타입은 LongWritable과 Text다. 따라서 만약 K2와 K3가 같다면 setMapOutputKeyClass()를 호출할 필요가 없다. setOutputKeyClass()에 정의된 타입으로 다시 설정되기 때문이다. 마찬가지로 V2와 V3가 같다면 setOutputValueClass()만 사용하면 된다.

중간과 최종 출력 타입을 설정하는 함수가 각각 존재하는 것이 이상하게 보일 수 있다. 왜 매퍼와 리듀서의 조합으로 이러한 타입을 결정할 수 없는 것일까? 해답은 자바 제네릭[1]의 한계와 관련이 있다. 즉, 타입 삭제type erasure[2] 때문에 런타임에 타입 정보가 항상 존재하지 않을 수 있으므로 타입 정보를 명시적으로 하둡에 제공해야 한다. 또한 컴파일 타임에 환경 설정을 점검하지 않기 때문에 맵리듀스 잡과 호환되지 않는 타입을 설정할 가능성도 있다. [표 8-1] 하단에 맵리듀스 타입과 반드시 호환되어야 하는 설정이 있다. 타입 충돌은 잡을 실행하는 런타임에 발견된다. 이러한 이유 때문에 먼저 작은 데이터로 테스트 잡을 실행하여 타입 호환성에 문제가 없는지 점검하는 것이 좋다.

**표 8-1** 새로운 API에서 맵리듀스 타입 설정

| 속성 | Job 설정 메서드 | 입력 타입 | | 중간 타입 | | 출력 타입 | |
|---|---|---|---|---|---|---|---|
| | | K1 | V1 | K2 | V2 | K3 | V3 |
| **타입 설정 속성:** | | | | | | | |
| mapreduce.job.inputformat.class | setInputFormatClass() | • | • | | | | |
| mapreduce.map.output.key.class | setMapOutputKeyClass() | | | • | | | |
| mapreduce.map.output.value.class | setMapOutputValueClass() | | | | • | | |
| mapreduce.job.output.key.class | setOutputKeyClass() | | | | | • | |
| mapreduce.job.output.value.class | setOutputValueClass() | | | | | | • |
| **타입과 반드시 일치해야 하는 속성:** | | | | | | | |
| mapreduce.job.map.class | setMapperClass() | • | • | • | • | | |

1 옮긴이_ 클래스에 사용할 타입을 디자인할 때 지정하는 것이 아니라 클래스를 사용할 때 지정하는 기법

2 옮긴이_ 타입 삭제는 JVM 레벨의 호환성을 위해 컴파일 타임에 제네릭 타입 정보(<T>, <V>, ...)를 삭제해버리는 기능이다.

| 속성 | Job 설정 메서드 | 입력 타입 | | 중간 타입 | | 출력 타입 | |
|---|---|---|---|---|---|---|---|
| | | K1 | V1 | K2 | V2 | K3 | V3 |
| mapreduce.job.combine.class | setCombinerClass() | | | • | • | | |
| mapreduce.job.partitioner.class | setPartitionerClass() | | | • | • | | |
| mapreduce.job.output.key.comparator.class | setSortComparatorClass() | | | • | | | |
| mapreduce.job.output.group.comparator.class | setGroupingComparatorClass() | | | • | | | |
| mapreduce.job.reduce.class | setReducerClass() | | | • | • | • | • |
| mapreduce.job.outputformat.class | setOutputFormatClass() | | | | | • | • |

표 8-2 이전 API에서 맵리듀스 타입 설정

| 속성 | Job 설정 메서드 | 입력 타입 | | 중간 타입 | | 출력 타입 | |
|---|---|---|---|---|---|---|---|
| | | K1 | V1 | K2 | V2 | K3 | V3 |
| **타입 설정 속성:** | | | | | | | |
| mapred.input.format.class | setInputFormat() | • | • | | | | |
| mapred.mapoutput.key.class | setMapOutputKeyClass() | | | • | | | |
| mapred.mapoutput.value.class | setMapOutputValueClass() | | | | • | | |
| mapred.output.key.class | setOutputKeyClass() | | | | | • | |
| mapred.output.value.class | setOutputValueClass() | | | | | | • |
| **타입과 반드시 일치해야 하는 속성:** | | | | | | | |
| mapred.mapper.class | setMapperClass() | • | • | • | • | | |
| mapred.map.runner.class | setMapRunnerClass() | • | • | • | • | | |
| mapred.combiner.class | setCombinerClass() | | | • | • | | |
| mapred.partitioner.class | setPartitionerClass() | | | • | • | | |
| mapred.output.key.comparator.class | setOutputKeyComparatorClass() | | | • | | | |
| mapred.output.value.groupfn.class | setOutputValueGroupingComparator() | | | • | | | |
| mapred.reducer.class | setReducerClass() | | | • | • | • | • |
| mapred.output.format.class | setOutputFormat() | | | | | • | • |

# [참고]API

2017년 11월 30일 목요일 오후 2:48

• org.apache.hadoop.mapred 패키지는 Hadoop mapreduce 를 위한 기본 Interface 및 mapreduce 를 위한 기본 부모 클래스를 다양하게 정의함.
• org.apache.hadoop.mapred.lib 패키지는 mapreduce 를 위한 hadoop에서 자체 구현한 class 들을 제공.

**Org.apache.hadoop.mapred 주요 interface**

| interface 명 | interface 설명 | 주요 Method 설명 |
|---|---|---|
| InputFormat | InputFormat 은 Map-reduce 잡의 Input의 여러가지 타입과 형태를 기술하게 지원. Map-reduce는 다양한 format 으로 input 을 받을 수 있는데(주로 Text 형) , 이렇게 input file 들이 특정 key,value 로 구성되었다면 KeyValueTextInputFormat 클래스로 지원 가능함.<br>Input으로 주어지는 파일들의 text format, key-value, 특정 length 고정, 특정 라인씩 무조건 분할 등 여러 조건으로 input file의 특성을 기재할 수 있다.<br>대표적인 구현 클래스는 TextInputFormat 임. | **InputSplit[] getSplits(JobConf job , int numSplits) throws IOException**<br>논리적으로 InputFormat으로 정의된 Input 들을 정해진 numSplits 개수만큼 분할함. 이때 논리적으로 분할된 InputSplit가 물리적인 Chunk 단위를 의미하지 않는다. |
| | | **RecordReader<K,V> getRecordReader(InputSplit split, JobConf job , Reporter reporter) throws IOException**<br>논리적으로 분할된 InputSplit를 Record 형식으로 접근하기 위한 Collection을 반환. RecordReader는 record의 boundary 를 규정하고 record를 보다 쉽게 접근할 수 있는 api를 제공함. |
| InputSplit | 개별 Mapper에 의해 처리되어야 할 분할 데이터를 나타냄. 각 Data Node 별로 존재 | |

| | | |
|---|---|---|
| | 하는 개별 Mapper에 원본 데이터를 분할하여 전달 되어야 함. 이렇게 분할 전달되는 입력 데이터의 기본 셋이 InputSplit 임. InputSplit 는 RecordReader 를 통해서 쉽게 record 기반의 데이터 처리 가능 | |
| Mapper | Input 으로 받은 key/value 쌍들을 reducer 로 전달하거나 또 다른 mapper 전달을 위한 중간 형태의 key/value 쌍으로 mapping 중요한 map 메소드를 정의함.<br>다양한 Mappling 형태를 지원하기 위해 hadoop에서는 Mapper Interface 를 여러가지 형태로 구현한 Class 제공.<br>IdentityMapper, InverseMapper, RegexMapper , TokenCountMapper 를 제공. | **Void map(K1 key , V1 value , OutputCollector<K2,V2> output , Reporter reporter) throws IOException**<br>K1, V1 으로 Input mapping 하여 OutputCollector 형태로 반환하는 함수. OutputCollector에 map의 일차 mapping 결과를 Collection 형태로 저장함. OutputCollector를 이용하여 map 결과 이용 가능. |

| | | |
|---|---|---|
| RecordReader | RecordReader 는 inputsplit 로 분할된 데이터를 Mapper와 Reducer에세 record 기반으로 쉽게 접근하기 위한 interface 를 제공.<br>Record의 boundary를 규정(Iterator와 유사하게 다음 record가 있는지로 검사 )하고 쉽게 Access 가능하게 해줌. | Boolean next(K key , V value) throws IOException<br>다음 key ,value 값을 읽어서 K , V 에 입력한다. Key/value가 읽히면 true 그렇지 않고 EOF 일경우 false; |
| | | **K createKey() :** Key 로 사용될 적절한 타입의 object 생성.<br>K createValue() : Value로 사용될 적절한 타입의 object생성. |
| Reducer | Reducer<K2,V2,K3,V3><br>Mapper로 부터의 output을 받아 동일한 key 를 가지는 값들을 하나로 묶는 중간 Set를 제공.<br>Shuffle 과 Sort 를 통해 동일한 key 값을 가지는 결과를 효율적인 병렬처리를 위해 최적화 함.<br>Reduce를 통해 동일한 key의 Value를 하나로 통합. | **Void reduce(K2 key , Iterator<V2> values , OutputCollector<K3, V3> output , Reporter report) throws IOException**<br>Map 의 결과 K2와 Iterator value 값을 받아 shuffle/sort/reduce 결과를 OutputCollector에 저장. |
| OutputFormat | 출력데이터를 파일형태로 편리하게 만들 수 있는 기능 제공을 위한 인터페이스. RecordWriter 객체를 반환할 수 있는 메소드를 제공하여 쉽게 파일 형태로 Write 가능하게 해줌. | **RecordWriter<K,V> getRecordWriter(FileSystem ignored, JobConf job, String name,Progressable progress) throws IOException**<br>해당 job 의 RecordWriter 객체를 반환함. Name 파라미터는 output 의 이름을 결정. |
| Partition | Map 에 중간 결과 partitioning을 조정할 수 있 | **Int getPartition( K2 key , V2** |

| | | |
|---|---|---|
| er | 는 기능을 제공. 일반적인 partitioning 방법은 key 값에 대한 hash 함수를 적용하는 것이다. Partitioning의 개수는 reduce task의 개수와 동일하다. | **value , int numPartitions)** Numpartitions에 주어진 총 partition개수 중에서 주어진 key에 따른 partition 번호를 반환한다. |

출처: http://barambunda.tistory.com/3 [바람이 분다]

출처: <http://barambunda.tistory.com/3>

# [참고]Hadoop 튜토리얼 (4-2) Map Reduce

2017년 12월 1일 금요일 오전 9:35

작성일: 2013년 4월 11일 글쓴이: admin

## MapReduce 의 데이터 흐름

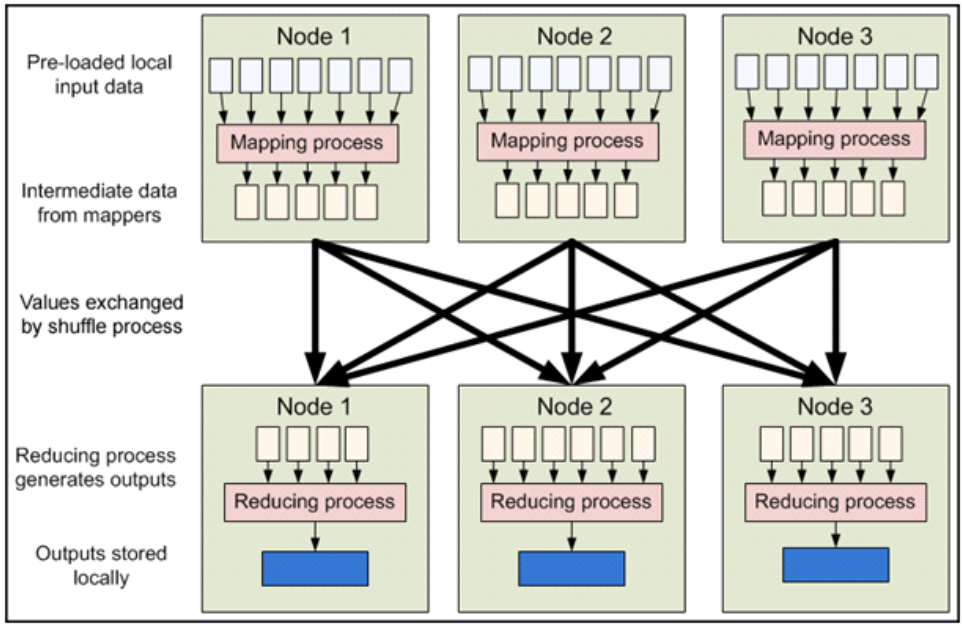



*Figure 4.4: High-level MapReduce의 pipeline*

HDFS 클러스터에 파일이 적재됨으로써 MapReduce 입력이 시작된다. 이들 파일은 전체 node에 균등하게 배분되는데 이에 대해 MapReduce 프로그램이 수행되면서 node에서는 mapping task가 시작된다. 이때 각각의 mapping task는 동등한 것으로서 이들 서로를 구별할 수 없으며 각 mapper는 그 어떤 입력파일도 처리할 수 있다. 각각의 mapper는 각 기기에 인접해 존재하는 파일들을 적재한 후 그 컴퓨터가 처리하게 한다.

mapping 단계가 끝나면 중간산출물로서의 intermediate (key, value) pair가 각 컴퓨터 사이에서 교환되고 같은 key를 가지는 모든 value들은 하나의 reducer에게 보내진다. mapper가 있는 node에는 reduce task들 역시 분포된다. **이러한 분배작업이 MapReduce에서 유일하게 통신이 이루어지는 부분이다.** 개별적인 map task는 상호간에 정보를 교환하지도 않을뿐더러 상호간의 존재조차 알지 못한다. 마찬가지로 각각의 reduce task들도 상호 간에 통신을 하지 않는다. 사용자 역시 기기간의 정보에 대해 명시적으로 marshal하지 않는다. 모든 데이터 전송은 Hadoop의 MapReduce 플랫폼에 의해 이루어지며 이때도 key를 중심으로 해당 value를 주고 받는 방식을 취한다. 바로 이런 점이 Hadoop MapReduce이 높은 신뢰성을 보장받는 가장 큰 이유이다. 클러스터 내의 node가 장애를 일으키면 task는 재수행된다. 만약 이들에게 소위 *side-*

*effects*를 수반된다면 (예: 외부와의 통신을 하고 있었다는 등) 그 상태정보 역시 재수행하는 새로운 task에 복구시킨다. 통신과 side-effect의 문제를 제거하고 차단함으로써 재시동은 자연스럽게 이루어질 수 있다.

## 세부 동작 원리

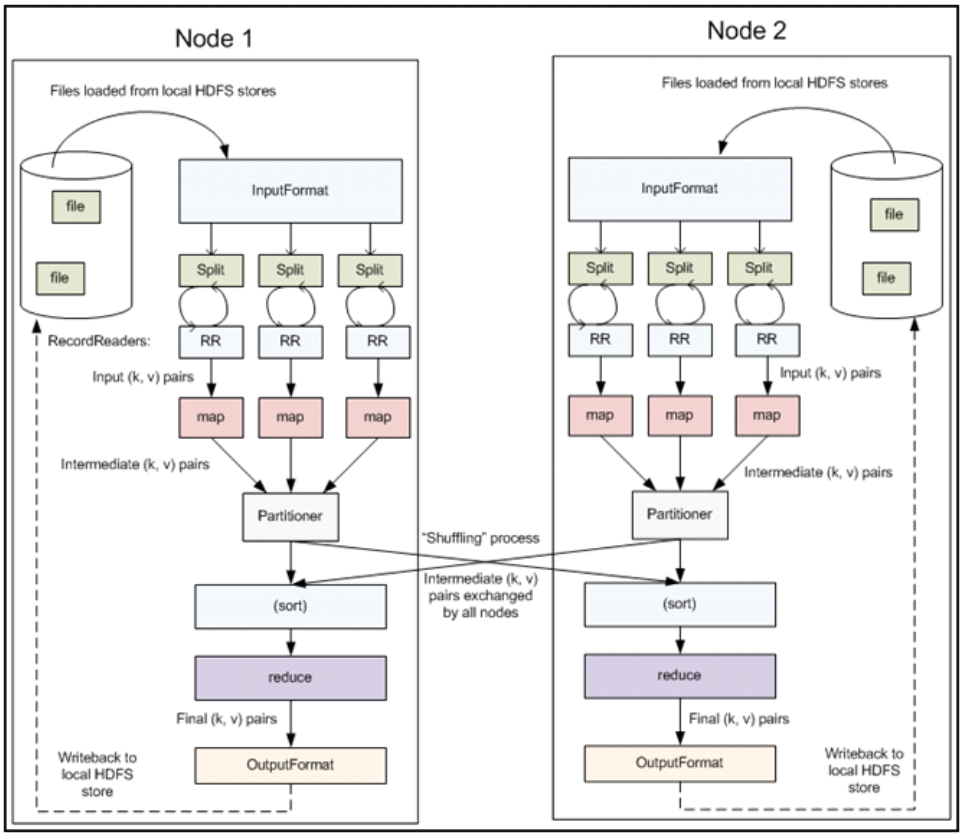



*Figure 4.5: Hadoop MapReduce의 데이터 흐름에 대한 상세한 모습*

Figure 4.5에서는 각 단계에서의 데이터 흐름을 pipeline으로 보여주고 있다.

**입력 파일:** MapReduce task를 위한 데이터가 저장되어 있는 곳으로서 보통 HDFS상에 존재할 것이다. 파일 포맷은 그때 그때 다르다: line기반의 log 파일도 있을 수 있고, binary 포맷, multiline의 입력 레코드 등 그 어떤 것도 가능하다. 다만, 용량은 매우 클 것이다.

**InputFormat:** 입력파일이 분할(split)되는 방식이나 읽어들이는 방식을 정의하는 class이다.

- 입력으로 사용될 파일 또는 기타의 object를 선택한다.
- 파일을 task로 분해할 *InputSplits* 을 정의한다.
- 파일을 읽어들이는 *RecordReader* 를 생성하는 factory를 제공한다.
  Hadoop은 여러 개의 InputFormat을 제공한다. *FileInputFormat* 라는 이름의abstract

type이 존재하며; 파일을 대상으로하는 모든 InputFormat은 이 class로부터 파생된다. Hadoop의 job을 수행하기 시작할 때 FileInputFormat에 읽으려고 하는 파일의 경로를 제시하면 FileInputFormat이 이 디렉토리에 있는 모든 파일을 읽어 들인다. 그런 후 각각의 파일을 여러 개의 InputSplit으로 분해한다. 개발자는 입력파일에 어떤 InputFormat을 적용할지에 대해서는 job을 정의하는 *JobConf* object의 setInputFormat() method를 호출하는 방식으로 정의한다. 표준의 InputFormat이 다음 표에 정의되어 있다.

| InputFormat: | 설명 | Key: | Value: |
|---|---|---|---|
| TextInputFormat | Default 포맷; 텍스트 파일의 각 line을 읽어들인다. | 각 line의 byte offset | 각 line의 내용 |
| KeyValueInputFormat | 각 line을 key, val pair로 parse한다. | 첫째 tab 문자까지의 모든 내용 | line의 나머지 내용 |
| SequenceFileInputFormat | Hadoop고유의 고성능 바이너리 포맷 | 사용자 정의 | 사용자 정의 |

Table 4.1: MapReduce가 제공하는 InputFormats

*TextInputFormat*은 디폴트의 InputFormat으로서 입력 파일의 각 line을 별개의 레코드로 취급하지만 별도의 parsing작업은 하지 않는다. 이것은 특히 unformatted 데이터 또는 log 파일과 같은 line기반의 레코드 등에 유용하다.

보다 흥미로운 입력 포맷은 *KeyValueInputFormat*으로서 이 역시 입력파일의 각각의 line을 별개의 레코드로 취급한다. 다만 TextInputFormat이 line 전체를 하나의 값(value)으로 여기는 반면 KeyValueInputFormat은 각 line을 tab 문자를 기준으로 key와 value로 분해한다는 점이 다를 뿐이다. 이는 특히 하나의 MapReduce job의 출력물을 읽어서 다른 MapReduce job의 입력항목으로 전달하는데 유용하다. 디폴트 상태에서의 OutputFormat (뒤에 설명) 이 그 결과를 이런 방식으로 포맷팅 하기 때문이다.

끝으로 *SequenceFileInputFormat* 은 Hadoop에 특수한 바이너리 파일을 읽어 들인다. 이 파일에는 Hadoop mapper에 고속으로 읽어들일 수 있도록 몇 가지 부가기능이 제공된다. Sequence 파일은 block-compressed 방식이고 (텍스트 이외에도). 몇 가지의 임의의 데이터 타입을 직접 serialization 및 deserialization할 수 있는 기능도 있다. Sequence 파일은 다른 MapReduce task의 출력물로 생성될 수도 있으며 한 MapReduce job에서 다른 MapReduce job으로 전달되는 임시 데이터의 표현형식이 되기도 하다.

**InputSplits:** InputSplit은 MapReduce 프로그램에서 *map task*를 구성하는 작업의

단위가 된다. Data set에 적용되는 MapReduce 프로그램은 이를 총체적으로 *Job*이라고 부르며 이 job은 여러 개의 (또는 수 백개의) task로 구성된다. Map task는 한 파일의 전체를 읽거나 일부분만을 읽을 수도도 있다. 디폴트 상태에서 FileInputFormat과 하위 class는 파일을 64 MB 단위의 chunk (HDFS에서의 블록 크기와 동일)로 분할한다. 이 값은 hadoop-site.xml에서 mapred.min.split.size 파라미터를 이용하거나 특정한 MapReduce job을 submit 시키는 기능을 가지는 JobConf object에서 파라미터를 override함으로써 변경할 수 있다. 파일을 chunk 단위로 처리하면 하나의 파일에 대해 여러 개의 map task를 병렬적으로 수행할 수 있게 된다. 파일이 매우 큰 경우 이러한 parallelism을 통해 성능을 획기적으로 높일 수도 있다. 더욱 중요한 것은 파일을 구성하는 다양한 블록을 클러스터 내의 여러 node에 분배할 수 있다는 것이다. 즉, 각각의 블록을 한 node에서 다른 node로 옮길 필요가 없이 해당 node에서 (locally) 처리할 수 있다는 것이다. 물론 log 파일과 같은 것은 이처럼 분할하여 처리가 가능하지만 어떤 파일포맷은 chunk단위로 쪼개기 어려울 수도 있다. 이때는 custom의 InputFormat을 작성해서 파일을 어떻게 분할할지를 (또는 일정조건에서는 분할되지 않도록 할지를) 상세히 지정할 수도 있다. Custom 입력포맷은 Module 5 에서 설명한다.

InputFormat은 mapping 단계에서의 각각의 task의 목록을 정의한다. 각각의 task는 각각 입력되는 split에 대응된다. 이들 task는 입력파일의 chunk가 물리적으로 어디에 위치하고 있는지를 기준으로 각 node에 할당된다. 개별 node마다 수십 개의 task가 할당된다. 각 node가 task에 대해 작업을 개시할 때는 가능한 많은 작업을 병렬처리하도록 노력한다. 이러한 각 node별 parallelism은 mapred.tasktracker.map.tasks.maximum 파라미터를 통해 통제된다.

**RecordReader:** InputSplit을 통해 일의 단위가 지정되었지만 이를 액세스하는 방법은 정의되지 않는다. 데이터를 source에서 실제로 적재한 후 이를 Mapper가 읽기에 수월한 (key, value) pair로 변환하는 일은 *RecordReader* class가 담당한다. RecordReader instance는 InputFormat에 의해 정의된다. 디폴트의 InputFormat인 *TextInputFormat* 은 *LineRecordReader*를 제공하는데 여기서는 입력파일의 각각의 line을 새로운 값(value)로 취급한다. 각 line에 대한 key는 파일에서의 byte offset이다. 입력 시 InputSplit 전체가 완료될 때까지 RecordReader는 계속 호출(invoke)된다. RecordReader가 호출되면 Mapper의 map() method 역시 호출된다.

**Mapper:** Mapper는 MapReduce 프로그램의 첫째 단계로서 사용자가 정의한 작업을 수행한다. Key와 value가 주어지면 map() method는 (key, value) pair(s)를 Reducer에게 전달한다. 새로운 Mapper instance는 각각의 map task (InputSplit)에 대한 별도의 Java 프로세스 속에서 만들어진다. 이러한 map task는 전체적으로 한의 job input을 구성한다. 각각의 mapper는 다른 mapper와 어떤 방식으로도 통신하지 않는데 이는 의도

적인 것이다. 이를 통해 각각의 map task의 신뢰성이 로컬 기기의 신뢰성에 전적으로 좌우되게 되는 것이다. map() method는 key와 value 이외에 2개의 파라미터를 전달받는다.

- *OutputCollector* object에는 collect() 라는 method가 있어서 (key, value) pair를 job의 reduce 단계로 전달해 준다.
- *Reporter* object 는 현재의 task에 대한 정보를 제공한다. Reporter의 getInputSplit() method는 현재의 InputSplit을 설명하는 object를 반환한다. 또한 map task로 하여금 진행상태에 대한 추가의 정보를 시스템 내 다른 요소에게 제공할 수도 있다. 또한 setStatus() method 를 통해 사용자에게 상태메시지를 제공할 수도 있다. incrCounter() method를 통해서는 shared performance counter를 증가시킬 수도 있다. 필요한 임의의 counter를 정의할 수 있으며 각 mapper는 counter를 증가시킬 수 있는데 JobTracker는 여러 프로세스에서 진행된 counter증가값을 수집한 후 이를 합계처리하여 job이 수행종료되었을 때 꺼내보게 된다.
  **Partition & Shuffle:** 첫번째 map task가 종료한 후에도 각 node들은 여러 개의 다른 map task들을 수행하고 있을 수도 있다. 그런 가운데서도 map task로부터의 중간산출물을 이를 필요로 하는 reducer에게로 전달하기 시작한다. 이처럼 map의 산출물을 reducer에게로 옮기는 것을 *shuffling*한다고 부른다. 중간단계 key space의 일부가 각각의 reduce node에 할당된다. 이들 subset (이를 “partition”이라고 부른다)들은 reduce task에게 입력된다. 각 map task는 그 어떤 partition에도 (key, value) pair를 전달할 수 있다. 하나의 key에 대한 모든 값은 항상 그 origin이 어떤 mapper였든 상관없이 병합(reduced together)된다. 따라서 중간산출 데이터의 각 항목을 어디로 보낼지에 대해 map node는 의견일치를 보아야 한다. *Partitioner* class는 주어진 (key, value) pair가 어떤 partition으로 갈지를 결정하는데 디폴트의 partitioner 는 key에 대한 hash 값을 계산한 후 그 결과에 따라 partition을 할당한다. Custom partitioner에 대해서는 Module 5 에서 설명.

  **Sort:** 각각의 reduce task는 여러 개의 중간 key에 관련된 value를 합산(reduce)한다. 개별 node에서의 일련의 중간 key는 Hadoop이 이를 자동으로 정렬한 후 Reducer에게 보내진다.

  **Reduce:** 각각의 reduce task에 대해 Reducer instance가 만들어진다. Reducer instance는 사용자가 제공하는 instance로서 job별로 중요한 2번째 단계가 된다. Reducer에게 할당된 partition에서의 각각의 key에 대해 Reducer의 reduce() method는 단 한번 호출되는데 이를 통해 key에 연결된 모든 value에 대한 iterator와 key를 받는다. iterator에 의해 하나의 key와 관련된 value들이 반환될 때 그 순서는 무작위이다. Reducer는 또한 *OutputCollector* 와 *Reporter* object를 파라미터의 형식으로 받게 되

는데 이들은 map() method에서와 같은 방식으로 이용된다.

**OutputFormat:** OutputCollector에게 제공되는 (key, value) pair는 출력파일에 기록된다. 실제 기록되는 방식은 *OutputFormat*에 의해 결정된다. OutputFormat 은 앞서의 InputFormat class 와 같은 방식으로 동작한다. Hadoop이 제공하는 OutputFormat의 instance는 로컬디스크 또는 HDFS상의 파일에 기록된다. 이들 모두 일반적인 *FileOutputFormat*에서 상속된 것이다. 각각의 Reducer는 각각의 파일을 일반적인 출력 디렉토리에 기록한다. 이들 파일은 통상 part-*nnnnn* 라는 이름을 가진다. (*nnnnn* 는 reduce task와 관련된 partition id이다.) 출력 디렉토리는 FileOutputFormat.setOutputPath() method에 의해 결정된다. 특별한 OutputFormat을 사용하려는 경우에는 MapReduce job을 정의하는 *JobConf* object 의 setOutputFormat() method를 통해 지정한다. 제공되는 OutputFormat은 다음과 같다.

| OutputFormat: | 설명 |
|---|---|
| TextOutputFormat | Default; line을 "key \t value" 형태로 기록한다 |
| SequenceFileOutputFormat | 뒤에 오는 MapReduce job으로 읽어 들이기에 적당한 형태의 바이너리 파일로 기록한다. |
| NullOutputFormat | 입력을 무시한다 |

Table 4.2: OutputFormats provided by Hadoop

Hadoop은 파일에 기록하기 위한 몇 가지 OutputFormat을 제공한다. 디폴트 상태의 instance는 TextOutputFormat으로서 텍스트파일의 각 line에 (key, value) pair를 기록한다. 이를 나중에 MapReduce task를 통해 *KeyValueInputFormat* class로 다시 읽어 들일 수 있는데 이는 사람도 읽을 수도 있다. MapReduce job들 상호간에 이용할 수 있는 더 좋은 중간 포맷이 *SequenceFileOutputFormat* 인데 이는 임의의 데이터 타입을 파일로 신속하게 serialize해 준다; 이에 대응되는 *SequenceFileInputFormat* 은 파일을 같은 타입으로 deserialize 하고 앞서의 Reducer가 산출했던 것과 같은 방식으로 다음 Mapper에게 전달한다. *NullOutputFormat* 은 아무런 출력파일을 만들지 않으며 OutputCollector에 의해 전달받은 (key, value) pair를 무시한다. 이는 reduce() method에서 독자의 출력파일에 기록하고 Hadoop 프레임워크에 의해 추가의 빈 출력파일이 만들지 않으려는 경우 유용하다.

**RecordWriter:** InputFormat이 실제로 개별 레코드를 RecordReader 실행을 통해 읽는 것과 마찬가지로 OutputFormat class도 *RecordWriter* object에 대한 factory역할을 한다. 이들은 OutputFormat에 지정된 대로.개별 레코드를 파일에 기록하는데 이용된다.

Reducer에 의해 작성된 **출력파일**은 HDFS에 남아있으므로 다른 MapReduce job 또는 별도의 프로그램 또는 사용자의 직접개입을 통해 이용할 수 있다.

추가적인 MapReduce 기능

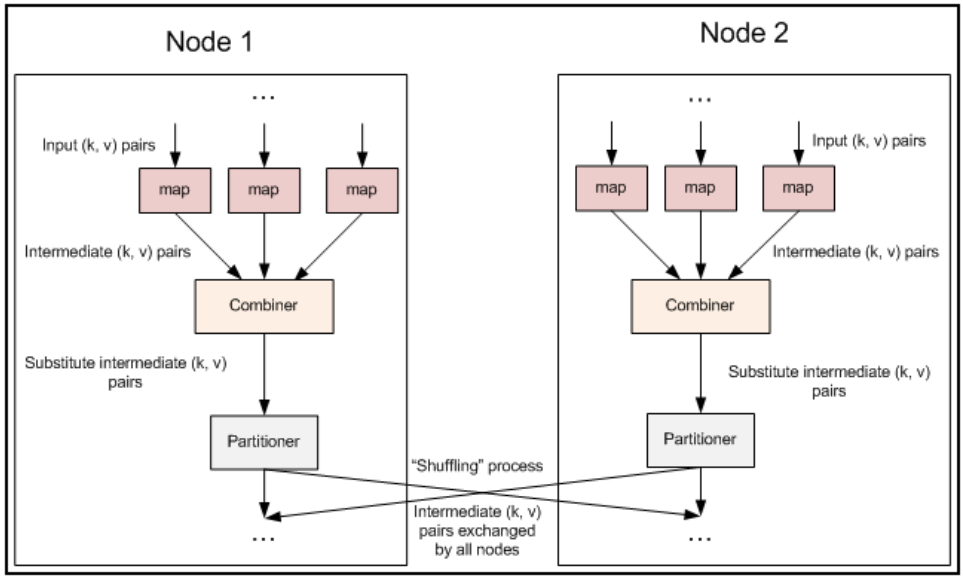



*Figure 4.6: Combiner step이 MapReduce의 데이터 흐름에 포함되었다.*

**Combiner:** 앞서의 그림에서 생략된 프로세스가 하나 있다. Combiner라고 불리우는 것인데 이것을 이용하면 MapReduce job이 사용하는 대역폭을 최적화하고 절감할 수 있게 된다. *Combiner* 는 Mapper와 Reducer사이에서 진행되는 것으로서 선택적으로 적용이 가능하다. Combiner를 적용하면 map task가 수행되는 모든 node에 대해 Combiner class의 instance가 적용된다. 각각의 node에서 Mapper instance가 산출한 데이터를 입력받고 Combiner가 지정된 작업을 하면 그 결과물을 Reducer에게 보낸다, Combiner는 일종의 **"mini-reduce" 프로세스로서 하나의 단위 컴퓨터에서 생성된 데이터만을 대상으로 한다.**

Word count프로그램이 대표적인 예가 될 수 있는데 listings 1–3에서 각 단어가 발견될 때마다 (*word*, 1) pair를 산출했다. 예컨대 "cat"라는 단어가 3번 발견되면 ("cat", 1) pair가 3번 출력되고 따라서 Reducer에게도 3번 전달이 되었다. 그러나, Combiner를 통해 이들이 단위 컴퓨터에서 합산되어서 ("cat", 3) pair가 단 한번만 Reducer에게 전달된다. 이처럼 모든 node 에서 여러 번 반복해서 전달되던 항목을 중간합산 하여 각 단어 당 한 번씩만 전달됨으로써 shuffle 프로세스에서 요구되는 대역폭을 획기적으로 줄일 수 있게 되고 결과적으로 job 처리속도가 개선된다. 무엇보다 좋은 점은 별도의 프로그래밍 작업을 할 필요 없다는 점이다. 만약 reduce 함수가 *commutative* 이고 동시에 *associative* 라면 이를 Combiner로도 이용할 수 있다. word count 프로그램의 경우 driver 프로그램에 다음 한 줄만 삽입하면 된다.

conf.setCombinerClass(Reduce.class);

Combiner는 *Reducer* interface의 instance여야 하지만 commutativity 또는 associativity 문제로 인해 Reducer 자체가 직접 Combiner로 이용될 수 없는 경우에도 Combiner로 사용할 제3의 class를 작성하면 가능하다.

이 글은 카테고리: Hadoop기술백서, 빅데이터에 포함되어 있으며 태그: hadoop distributed file system, Hadoop기술백서, HDFS, mapper, mapreduce, marshall, Moore의 법칙, reducer, 분산컴퓨팅, 분산파일시스템 (이)가 사용되었습니다. 고유주소를 북마크하세요.

← Hadoop 튜토리얼 (4-1) Map Reduce

출처: <http://www.openwith.net/?p=506>

# [참고]Hadoop 아키텍처

2017년 12월 1일 금요일 오전 9:38

작성일: 2013년 5월 27일 글쓴이: admin

## (1) 개요

Hadoop은 빅데이터 처리를 위해 여러 대의 컴퓨터로 클러스터를 구성하는데 클러스터란 특정한 기능수행을 위해 여러 대의 컴퓨터가 네트워크로 연결한 것을 말하며 이때 클러스터를 구성하는 개별 컴퓨터를 노드(node)라고 한다. (이하에서는 node라 칭한다)

Hadoop은 이처럼 물리적으로는 node를 연결하고 그 위에서 다양한 기법을 동원하여 거대한 프레임워크를 구축하였다. 이하에서 Hadoop의 아키텍처를 다양한 측면에서 살펴 본다.

### 사용자 차원

사용자 차원에서는 분석가가 분석 모델링을 한 후 프로그래머에게 프로그램 작성을 요청한다. 프로그래머는 Hadoop 프레임워크에 맞추어 프로그램을 작성한 후 이를 실행한다. 한편 클러스터 환경에서의 데이터와 시스템의 관리는 컴퓨터 운영관리자가 담당한다. 분석가는 실제 분석작업을 진행하여 사업상의 중요한 정보(Intelligence 내지 Insight)를 찾아낸다. 만약 복잡한 모델링이 필요치 않은 일상적 분석의 경우에는 Pig 및 Pig Latin과 같은 Hadoop 내 하위 프로젝트에서 제공하는 간편한 질의어를 이용하기도 한다. 이 내용이 다음 그림에 잘 표현되어 있다.[1]

### 물리적 차원

Hadoop 클러스터는 한 대의 네임노드 (NameNode)서버와 여러 대의 데이터노드 (DataNode)서버로 구성된다.

### 논리적 차원

논리적으로는 Hadoop 프레임워크는 여러 개의 데몬 (daemon) 프로그램으로 구성된다. 데몬 프로그램이란 서버의 메인메모리 상에서 백그라운드로[2] 수행되는 여러 가

지 프로그램을 말한다. 이때 여러 개의 데몬 프로그램이 상호 동작하는 방식으로서 Hadoop은 master-slave의 형태를 취하는데 Master란 작업을 관리하고 Slave는 Master의 관리하에 실제 작업을 진행하는 것을 말한다.

기능적 차원

Hadoop은 핵심모듈(core module)과 관련 프로젝트로 구성되며 핵심모듈은 다음의 몇 가지 요소로 구성된 된다.

n HDFS (Hadoop Distributed Filesystem) – Haddop의 파일시스템으로서 데이터를 저장하고 이를 Hadoop 프레임워크에 입력하는 역할을 한다.

n MapReduce – Hadoop의 대표적인 프로그래밍 모델이다.

n 기타의 데몬 프로그램 –HDFS와 MapReduce를 보조하여 전체적인 운영효율을 높여주는 각종의 유틸리티 소프트웨어이다.

결국 Hadoop 프레임워크는 주로 (데이터 관리 layer로서의) HDFS와 (프로그래밍 모델로서의) MapReduce의 2가지로 구성된다. 그리고 이들 (HDFS와 MapReduce) 모두 각각 master기능을 하는 데몬 프로그램과 slave 역할을 하는 데몬 프로그램으로 나뉘어진다. 즉, Hadoop에서의 서버 node에서는 각각 다음의 여러 가지 데몬(daemon) 프로세스[3]를 수행한다.

n NameNode (및 Secondary NameNode)

n DataNode

n JobTracker

n TaskTracker

이중 NameNode와 DataNode는 HDFS에 대한 기능을 하는 데몬 프로그램인 반면 JobTracker와 TaskTracker 데몬 프로그램은 MapReduce에 대한 기능을 하는 데몬이다.

그리고 NameNode (및 Secondary NameNode)와 JobTracker는 master node에서만 수행되는 반면 JobTracker와 TaskTracker 데몬 프로그램은 나머지 여러 대의 slave node에서 수행되게 된다.

| | **Master node (서버)** | **Slave node (서버)** | **클라이언트 PC (작업자)** |
|---|---|---|---|
| **HDFS** | **NameNode** (Secondary node가 있을 수 있다) | **DataNode** | Master node에게 지시하고 진행상황을 출력한다. |
| **MapReduce** | **JobTracker** | **TaskTracker** | |

달리 말하면 Hadoop은 논리적 아키텍쳐 상에서 "저장을 위한 계층 (HDFS layer)"과 "실제 작업을 위한 계층 (Computation layer 즉, MapReduce layer)"의 2개 레이어로 구성된다. 이들 각각에서는 해당되는 여러 데몬 프로그램이 수행되면서 사용자의 명령을 기다린다.

명령은 명령어 줄(command-line)을 이용할 수도 있으나 주로 알고리즘 중심의 프로그래밍 (Data Flow Programming)방식을 이용하는 것이 일반적이다.

## (2) Hadoop의 처리 흐름도

Hadoop에서는 데이터가 그 중심 역할을 한다. 따라서 여기서는 데이터와 그 처리 흐름을 중심으로 살펴 본다.

MapReduce에서:

n 하나의 master node (JobTracker)와 여러 개의 작업 node (TaskTrackers)를 운용

한다.

n 클라이언트 PC는 job울 JobTracker에게 submit한다

n JobTracker는 각각의 job을 tasks로 분할한다 (map/reduce)

n 필요에 따라 task를 TaskTrackers 에게 배분한다.

HDFS(Hadoop Distributed File System)에서

n 하나의 name node와 여러 개의 data node를 운용한다.

n 데이터는 고정길이(64 MB)의 블록으로 나누어서 처리한다

n HDFS에서는 입력작업을 map 처리하고 출력작업을 reduce 한다.

[1] source: KarmaSphere

[2] 백그라운드 작업(background job) 이란 이용자의 특별한 작업 없이 배후에서 시스템이 자체적으로 수행하는 프로세스를 말한다.

[3] 프로세스란 프로그램이 수행되고 있는 상태를 말한다. 데몬 프로세스는 이 중에서 사용자가 별도의 수행명령을 내리는지 여부에 상관없이 (메인메모리 상에서) 항상 수행되면서 대기하고 있는 것을 말한다. 메모리 상주 프로그램 또는 서비스 프로그램이라고도 한다.

출처: <http://www.openwith.net/?p=547>

# [참고]Hadoop의 기능적 요소 – HDFS와 MapReduce

2017년 12월 1일 금요일 오전 9:38

## (1) HDFS (Hadoop Distributed File System)

### ① HDFS 설계원칙

데이터를 많은 사용자가 네트워크 환경에서 이용하기 위해 개발된 분산파일시스템으로는 NFS (Network File System)가 대표적이다. 그러나 NFS는 하나의 기기에 보관된 하나의 논리볼륨 (logical volume) 만을 원격에서 액세스한다는 제한점을 가진다. HDFS는 이러한 NFS의 한계를 극복하기 위해 고안되었고 설계원칙 상 다음과 같은 특징을 가진다.

n HDFS는 대용량 데이터 (terabyte 또는 petabyte)를 저장하도록 고안되었다. 이를 위해서 데이터를 여러 대의 컴퓨터에 나누어 저장시킨다. 즉, NFS보다 훨씬 큰 파일도 지원한다.

n HDFS에서 데이터 저장의 신뢰성이 훨씬 높아서 개별 컴퓨터가 이상을 일으켜도 데이터를 이용할 수 있다.

n HDFS는 데이터 액세스에 대한 성능개선도 용이하다. 클러스터에 컴퓨터 node만 추가하면 이용할 수 있는 클라이언트 숫자는 계속 늘어날 수 있다.

n HDFS는 Hadoop의 MapReduce와 잘 통합된다.

반면 HDFS의 성능은 개선되었지만 설계원칙 상 불가피하게 특정 응용프로그램에는 적합하지 않은 결과를 초래한다. 즉, NFS만큼 범용은 아니라는 뜻이다. HDFS를 이용 시 다음의 상충되는 요소들을 함께 고려해야 한다.

n HDFS는 파일을 순차적 스트리밍 방식으로 읽는 (long sequential streaming reads) 응용프로그램을 전제로 하였다. 따라서 무작위로 random access하는 경우에는 탐색시간 (seek times)이 길어지는 단점이 있다.

n HDFS는 한번 기록한 후 읽기를 여러 번 반복하는 것(write-once, read-many)을 기준으로 하였다. (파일에 데이터를 추가하는 기능이 Hadoop 0.19부터 지원되기는 하지만 성능상 한계가 있다.)

n 작업 대상 파일의 크기가 크고 읽기 작업이 순차적으로 이루어지다 보니 HDFS에서는 데이터의 캐싱 (local caching)을 하지 않는다. 실제로 캐싱작업에서는 시스템의 부담이 커지므로 차라리 데이터의 소스로부터 읽기작업을 다시 실행하는 것이 오히려 더 나은 경우가 많다.

n 개별 기기가 장애를 일으키는 일이 빈번할 뿐 아니라 여러 대의 기기가 동시에 문제를 발생시키는 경우도 있으므로 (예: rack 의 장애로 인해 여러 대 컴퓨터가 동시에 동작하지 않는 경우) 클러스터는 이러한 것들을 견딜 수 있도록 해야 한다. 이 때 비록 일부 기기의 성능이 저하되는 한이 있더라도 시스템 전체가 느려지거나 데이터를 이용할 수 없는 결과가 초래되지 않도록 대책을 세워야 한다.

당초 HDFS는 GFS (Google File System)을 기초로 설계된 블록 기반의 (block-structured) 파일시스템이다. 따라서 각각의 파일을 일정한 크기의 블록(block)으로 나눈 후 클러스터 내의 여러 기기에 분산 저장하는데 이때의 개별 기기에는 DataNode라는 파일시스템 데몬이 동작한다. 이때 Hadoop은 이들 각각의 블록을 보관하는 컴퓨터를 블록 별로 (on a block-by-block basis) 무작위로 선택한다. 결과적으로 파일을 이용하는 데에는 여러 컴퓨터 사이의 협력이 필요해지는 대신 단일기기용 분산파일시스템보다는 지원하는 파일의 크기가 훨씬 커지는 장점이 있다.

파일을 이용하는데 여러 대 컴퓨터가 협력해야 한다는 점에서는 이들 중 한 대만 고장을 일으켜도 파일작업을 할 수 없게 된다는 문제가 발생하는데 HDFS에서는 각각의 블록을 여러 대의 컴퓨터에 중복하여 저장하는 (이때의 중복 저장하는 숫자를 replication factor라고 하며 3을 기본으로 하되 조정이 가능하다) 상식으로 이 문제를 해결한다.

DataNode들이 여러 파일의 블록들을 저장하고 있다. 이 그림에서는 replication factor가 2로 되어있다. NameNode는 파일이름과 그 파일의 block id를 대응시킨다.

대부분의 파일시스템에서 블록의 크기는 4 또는 8 KB이다. 반면 HDFS에서의 기본 (default) 블록사이즈는 64MB로서 훨씬 큰데 이로 인해 파일 당 필요한 메타데이터는 작아지는 결과가 초래된다 (파일 당 필요한 블록의 개수 자체가 줄어든다).

또한 HDFS에서는 데이터를 순차방식으로 고속 stream read하므로 대용량 데이터를 효율적으로 읽어 들일 수 있다. NTFS 또는 EXT 같은 일반 파일시스템에서는 파일의

평균 크기가 작다고 전제하고 있지만 HDFS는 크기가 큰 파일을 전제로 한다. 예컨대 100MB 크기의 파일이라고 해도 블록 2개면 충분하다. PC의 경우 파일 크기도 작고 이곳 저곳의 내용을 random하게 액세스하는 경우가 대부분이지만 HDFS는 프로그램이 블록을 처음부터 끝까지 읽어 들인다고 가정하였으며 이로 인해 MapReduce 스타일의 프로그래밍이 편리해진다.

HDFS에서의 데이터 파일을 처리하는 주된 기준은 레코드 단위이다. 즉, 입력파일을 레코드를 기준으로 절단(split)하며 각 프로세스는HDFS 파일의 위치에 따라 (이를 지역성(locality)이라 한다) 할당된 record만 처리하게 되는 것이다. 이는 일견 당연해 보이지만 실제로는 기존 데이터처리 방식과는 정 반대이다. 즉, 기존에는 작업대상 파일(즉, 스토리지)을 전산시스템의 중앙부에 놓고 – 이때 관리효율을 높이기 위해 SAN (Storage Area Network) 내지 NAS (Network Attached Storage)를 이용함 – 모든 컴퓨터 node는 이들 중앙의 데이터 중 필요한 부분을 가져와서 작업하고 결과를 다시 중앙 스토리지에 반환/저장하는 방식을 취하였다. 그러나 과정을 유기적으로 조정하기 위해 많은 통신 오버헤드 – 예컨대 socket 프로그램 내지 응용프로그램에서의 marshalling 등- 가 과도하게 발생하였다.

그러나 Hadoop에서는 데이터를 중앙에 모으는 대신 이를 처리할 컴퓨터에게 보내준다[1]는 (Moving computation to the data) 발상의 전환을 통해 기존방식에서의 관리부담 (overhead)이 원천적으로 사라지게 되었다.

Hadoop의 프로그래밍 프레임워크에서 데이터는 논리적 차원에서 레코드 단위로 처리(record-oriented)된다. 따라서 각각의 입력파일은 여러 개의 줄(line) 또는 별도로 응용프로그램의 로직에 의해 지정된 나름의 형식으로 분해된다. 클러스터 내의 각 node에서 수행되는 프로세스들 역시 이들 레코드 단위로 작업을 수행한다.

구체적으로는 대상 데이터 세트(dataset)[2]을 블록단위로 나누고 중복적으로 slave node에게 분배하는 한편 이들에 대한 MapReduce의 작업 결과를 읽거나 파일에 기록하는 일련의 파일작업을 포함한다. 다만, HDFS는 일반적인 Linux 의 파일시스템과는 독립된 것이므로 반드시 Hadoop 명령어를 통해서만 이를 관리할 수 있다.

일반적으로 Hadoop에서는 데이터파일을 Linux 파일로 생성한 후 이를 HDFS로 복사해 와서 작업하는 방식을 취한다. 그리고 그 파일 대해서 응용프로그램을 통해 MapReduce 작업을 수행할 때도 모든 작업은 HDFS 파일을 직접 다루기 보다는 MapReduce를 통해 (key/value)의 쌍으로 이루어진 record단위로 작업을 수행하는 방식을 취한다. 즉, 대부분의 경우 HDFS 파일에 대해 직접 기록하거나 읽는 대신 응용 프로그램에서의 MapReduce를 통해 파일작업이 진행되는 것이다.

② HDFS 관련 데몬

NameNode

HDFS에서의 master인 NameNode는 분산환경에서 저장기능을 담당한다. 즉, 실제 작업의 대상이 되는 파일을 블록(block)단위로 나누어서 slave node들에게 분배할 뿐만 아니라 전체적인 (분산) 파일시스템의 이상 유무도 체크하고 slave 컴퓨터인 DataNode에서의 데이터 입출력 작업 (low-level I/O tasks)을 지휘한다. 이러한 작업은 메모리 소모도 크고 입출력도 많이 일어나므로 NameNode에 이상이 발생하면 Hadoop 클러스터는 전체적으로 동작을 멈추게 된다.

파일시스템에서 메타데이터 관리의 신뢰성도 중요하다. 특히 데이터 파일은 "write once, read many" 형태를 취하지만 메타데이터 (예: 파일 및 디렉토리 이름 등)만큼은 여러 클라이언트 컴퓨터가 동시에 수정을 시도할 수 있으므로 동기화 기능이 중요하기 때문에 NameNode가 별도의 컴퓨터에서 관리한다. NameNode는 파일시스템의 모든 메타데이터를 관리하는데 파일당 메타데이터의 크기가 작으므로 (파일명, 사용권한, 각각의 블록의 위치 정도만 관리) 이들 메타정보는 NameNode 기기의 메인메모리에 상주시켜 이용한다. 이처럼 한 클러스터에는 단 한 개의 NameNode가 존재하고 master node로 지정된 컴퓨터가 NameNode를 전담관리하며 다른 작업은 일체 하지 않는다.

클라이언트는 NameNode에게 질의하여 메타데이터 즉, 특정 파일의 블럭의 목록 등은 NameNode를 통해 메인메모리로부터 가져오지만 이후의 작업은 NameNode의 간섭 없이 DataNode로부터 병렬로 직접 read 작업을 수행한다. 이처럼 NameNode의 관여 없이 데이터를 통째로 가져오기 때문에 효율이 높아진다.

개별 DataNode가 장애를 일으키는 경우에도 전체적인 시스템의 이용은 유지되지만 NameNode의 장애는 시스템 전체의 이용이 불가능한 상태를 초래한다. 다만 일상적인 작업현장에서 DataNode와는 달리 NameNode는 주도적 역할을 하지 않기 때문에 장애의 위험성은 훨씬 적다. 만전을 기하기 위해 NameNode를 이중으로 가져가려는 노력이 있을 수 있는데 그 방법이 secondary NameNode의 이용이다.[3]

DataNode

Hadoop cluster에서는 데이터를 읽어 들이는 즉시 각각의 node에 데이터가 분배된다. HDFS는 큰 데이터 파일을 여러 개로 분리시켜서 각각의 node가 이를 처리하게 하는데 이들 각각의 조각(chunk) 는 여러 대의 컴퓨터에 중복적으로 복제되어서 한 컴퓨터에서 장애가 발생해도 다른 컴퓨터를 통해 데이터를 이용할 수 있다. 또한 모니터링 시스템을 지정해서 저장된 데이터의 일부에 문제가 발견될 때 이를 다시 복제하도록 조처

한다. 이때 이들 모든 파일조각들은 하나의 namespace를 공유하므로 클러스터 내의 모든 node들은 이를 이용할 수 있다. slave 기기는 DataNode daemon[4]을 통해 분산파일의 read/write 작업을 수행한다. 그리고 이때 DataNode 데몬 프로그램은 다른 DataNode와 통신하면서 자신의 데이터 블럭을 복제하기도 하고 직접 처리작업을 수행한다.

실제로 HDFS의 분산파일시스템을 대상으로 한 읽기 및 쓰기의 모든 작업이 이루어진다. 모든 작업은 대상 파일을 random하게 블록(block) 단위로 나누어 진행하는데 이들 DataNode의 작업은 수시로 NameNode에 보고되고 그 내역은 NameNode에 메타데이터의 형식으로 저장된다.

③ HDFS 명령어

HDFS는 일반 Unix/Linux의 파일시스템과는 전혀 별개이다. DataNode 데몬을 수행하는 기기에서 ls 명령을 수행하면 일반 Linux 파일시스템의 내용은 보이지만 HDFS의 파일은 보이지 않는다. 마찬가지로 fopen() 또는 fread()같은 표준의 읽기/쓰기 작업도 불가능하다. 요컨대 파일시스템으로서의 각종 작업에 대해서 HDFS는 HDFS 나름의 독자적인 명령어와 shell 구조를 가지고 있다는 말이다. 이러한 현상은 HDFS가 별도의 독립된 namespace,를 가지기 때문이다. HDFS (정확히는 HDFS를 구성하는 블록) 내의 파일은 DataNode 서비스가 관리하는 별도의 디렉토리에 저장된다. 그리고 이들 파일은 block id로만 표시된다. HDFS에 저장된 파일에 Linux의 일반 파일수정방식을 (예: ls, cp, mv, etc) 적용할 수도 없다. 대신 HDFS는 자체의 파일관리 방식을 가지는데 그 모습은 기존의 방식과 매우 유사하다.

한편 HDFS 파일시스템을 이용하려면 처음 한 차례에 한해 다음 HDFS 명령어를 통해 포맷팅이 되어야 한다.

user@namenode:hadoop$ bin/hadoop namenode -format

이하에서는 이러한 선행작업이 되었다고 전제한다.

한편 Hadoop의 작업에서는 일반적으로 데이터 파일이 Hadoop 이외의 곳에서 생성되

는 것을 전제로 하는 경우가 많다. 즉, 별도의 텍스트 파일이 있다거나, 또는 각종의 log 파일이 외부에서 이미 생성되었다고 보고 이를 HDFS로 복사하는 방식으로 읽어 들이는 것이다. 그리고 파일을 HDFS로 읽어 들인 후에도 MapReduce 프로그램이 이를 처리할 때는 MapReduce 응용프로그램에서 직접 읽는 대신 Hadoop의 MapReduce 프레임워크를 통해 HDFS 파일을 (key/value pair형태의) 개별 레코드 형식으로 파싱(parsing)하여 사용하게 된다.

### 기본적인 파일 shell 명령어

Hadoop의 파일 명령어의 표준형은 다음과 같다.

Hadoop fs –cmd <*args*>

여기서 cmd는 구체적인 명령어를 그리고 <args>는 매개변수(argument)를 나타낸다.

다만 명령어 cmd는 일반 Unix/Linux의 명령어와 그 형태가 매우 유사하다. 예를 들어서 특정 디렉토리 내에서의 파일의 목록을 보려면:

Hadoop fs –ls

라고 하면 된다.

HDFS의 디폴트 작업 디렉토리는 /usr/$USER 이지만 (단, $USER는 login 한 사용자 이름을 뜻함) 그렇다고 그 디렉토리가 자동으로 만들어지지는 않으므로 다음의 명령어를 통해 이를 직접 만들어 주어야 한다.

Hadoop fs –mkdir /user/hkyoon

Linux/Unix 상에 존재하는 로컬파일을 HDFS로 복사해 오는 명령어는 다음과 같다.

Hadoop fs –put example.txt

앞서 ls 명령을 재귀적으로 사용하여 하위 디렉토리 내의 목록까지 함께 보려면 lsr 명령어를 이용한다.

$ hadoop fs -lsr /

drwxr-xr-x – hkyoon supergroup 0 2013-01-14 10:23 /user

drwxr-xr-x – hkyoon supergroup 0 2013-01-14 11:02 /user/hkyoon

-rw-r–r– 1 hkyoon supergroup 264 2013-01-14 11:02 /user/hkyoon/example.txt

위 목록에서 마지막 줄의 소유자 이름 (hkyoon) 앞의 1이란 숫자가 복제본 개수 (replication factor)를 나타낸다.

앞서의 put과 반대로 HDFS로부터 로컬시스템의 파일로 가져오는 명령으로 get이 있다. 한편 파일의 내용을 보려면 cat이라는 명령어를 사용한다. Unix/Linux의 catalog에 해당한다. 다음에서는 여기에 head라는 Linux 명령을 pipeline으로 연결하여 적용했다.

hadoop fs -cat example.txt | head

이외에도 다음과 같은 다양한 HDFS의 파일명령어가 제공된다.

| | | | |
|---|---|---|---|
| cat | chgrp | chmod | chown |
| copyFromLocal | copyToLocal | cp | du |
| dus | expunge | get | getmerge |
| ls | lsr | mkdir | movefromLocal |
| mv | put | rm | Rmr |
| setrep | Stat | tail | test |
| text | touchz | | |

이들의 용법에 대해서는 http://hadoop.apache.org/docs/r0.18.3/hdfs_shell.html 참조.

프로그램을 이용한 HDFS 파일의 이용

Hadoop에서는 파일열기(open), 읽기(read), 기록하기 (write), 파일 닫기(close) 등을 비롯해 각종의 작업을 할 수 있는 HDFS API가 제공되므로 이를 이용할 수 있다. 이를 위해서는 우선 org.apache.hadoop.fs 의 package와 함께 필요한 API를 프로그램 내에서 이용하는데 여기서는 세부 내용을 생략한다.

hadoop이 제공하는 다양한 api는 apache 내 hadoop 사이트의 관련 문건을 보면 된다.
( http://hadoop.apache.org/docs/current/api/ )

아래에서는 hadoop.txt라는 이름의 파일을 만든 후 "My First Hadoop API call!\n"라는 문자열을 기록하고 다시 이를 읽어서 화면에 출력하는 프로그램이다. (전체적 흐름에 대한 이해를 돕기 위해 주석 및 package import 부분은 생략하였음)

```
public class HDFSExample {

public static final String FileName = "hadoop.txt";

public static final String message = "My First Hadoop API call!\n";


public static void main (String [] args) throws IOException {

Configuration conf = new Configuration();

FileSystem fs = FileSystem.get(conf);

Path filenamePath = new Path(theFilename);
```

```
try {
if (fs.exists(filenamePath)) {
fs.delete(filenamePath);
}

FSDataOutputStream out = fs.create(filenamePath);
out.writeUTF(message);
out.close();

//Open Config file to read
FSDataInputStream in = fs.open(filenamePath);
String messageIn = in.readUTF();
System.out.print(messageIn);
in.close();
} catch (IOException ioe) {
System.err.println("IOException during operation: "
+ ioe.toString());
System.exit(1);
}
}
}
```

## (2) MapReduce

①MapReduce 기본개념

MapReduce는 Hadoop의 프로그래밍 패러다임으로서 입력되는 데이터 리스트(list)를 출력 리스트(list)로 변형(transform)시킴에 있어 과정 전체를 한번에 처리하지 않고 크게 두 덩어리로 나눈 후 한 단계의 작업이 완료되면 그 다음의 단계로 이어지도록 하는 것을 말한다. 마치 Unix/Linux에서의 pipeline이 여러 개의 명령어를 연결시켜서 차례로 수행하도록 하는 것처럼 MapReduce 프로그램은 데이터 처리작업을 Map과reduce라는 두 번에 걸쳐 실시한다. 그리고 이 과정에서 partitioning과 shuffling 작업이 보조적으로 수행된다.

그림: 입력 list의 각 항목에 Mapping 함수를 적용되어 새로운 출력 list가 만들어진다.

프로그래머는 mapper와 reducer 에 자신의 필요한 작업내용을 프로그램 형태로 지정한 후 Hadoop 에서 수행시키면 이들 mapper와 reducer가 partitioning 및 shuffling의 지원을 받아가면서 분산 처리되는 것이다. 구체적으로는:

mapper 단계에서 먼저 입력 파일을 분할하여 여러 node에 (중복) 배분한 후 각각 할당된 데이터를 처리한다.

Reducer는 mapper의 처리결과를 넘겨받은 후 이들을 종합하여 최종결과물을 만들어내다.

그림: 색깔 별로 각각의 key를 나타낸다고 가정하였다. 결국 같은 key를 가진 value는 하나로 통합된다.

다음의 예에서는 몇 가지 단어가 수록된 입력파일을 분석해서 단어별 빈도수를 조사하는 모습을 그림으로 표현하였다.

Haddop의 데이터 타입

MapReduce에서는 사용자가 임의로 타입을 지정하거나 또는 Java의 기본 데이터 타입

을 그대로 이용하는 것을 허용하지 않으므로 반드시 MapReduce 프레임워크가 지원하는 데이터 타입을 이용해야 한다. Hadoop에서 key 와 value에 대해 적용할 수 있는 데이터 타입의 규칙은 다음과 같다.

n Writable interface를 실행하는 class는 value가 될 수 있다.

n WritableComparable<T> interface를 실행하는 class는 key 또는 value 모두가 될 수 있다.

n (참고로 WritableComparable<T> interface는

n Writable과 java.lang.Comparable<T> interface를 합한 것이다.

n 이처럼 key를 정의할 때 비교가능성(comparability)가 필요한 것은 value와는 달리 이들이 reduce 단계에서 정렬되기 때문이다.

다음은 Hadoop이 기본으로 제공하는 데이터 타입인데 이들은 Java 표준 데이터타입에 대한 wrapper로서 WritableComparable interface를 구현하며 key/value pair로 많이 이용된다.

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

물론 이 밖에 필요에 따라 위의 interface를 구현하여 자신만의 데이터타입을 만들어 사용할 수 있다.

Mapper 함수와 Reducer 함수

Mapper로 사용하기 위해서 해당 class를 작성할 때는 Mapper interface를 구현하면서 MapReduce base class를 확장해야 한다. MapReduce class는 mapper 및 reducer 모두의 base class로서 다음의 2개 method를 가진다.

n void configure(JobConf job)

n void close( )

Mapper interface는 map() 이라는 단 하나의 method를 가지는데 그 signature는 다음

과 같다.

void map(K1 key,

V1 value,

OutputCollector<K2,V2> output,

Reporter reporter

) throws IOException

다음은 Hadoop에서 미리 제공하는 Mapper의 실행 class이다. (각각의 의미는 http://hadoop.apache.org/docs 참조.)

| | |
|---|---|
| IdentityMapper<K,V> | InverseMapper<K,V> |
| RegexMapper<K> | TokenCounterMapper<K> |

Reducer 역시 MapReduce base classs를 확장하면서 Reducer interface를 구현해야 한다. Reducer interface 역시 reduce() method를 가지며 그 signature는 다음과 같다.

void reduce(K2 key,

Iterator<V2> values,

OutputCollector<K3,V3> output,

Reporter reporter

) throws IOException

다음은 Hadoop에서 미리 제공하는 Reducer의 실행 class이다.

(http://hadoop.apache.org/docs 참조.)

| IdentityReducer<K,V> | LongSumReducer<K> |
|---|---|

Word Count의 예

이제 조금 구체적인 예제 프로그램을 살펴 본다. 파일의 내용을 분석하여 출현하는 단어 별 빈도를 출력하는 것으로서 Hadoop을 설치하면 함께 오며 위치는 src/examples/org/apache/hadoop/examples/WordCount.java 이다. 다만, 여기서는 한글 데이터 파일을 대상으로 하는 것으로 가정하였고 프로그램은 대폭 단순화 하였다.

우선 다음 시(詩)를 내용으로 하는 파일이 있다고 가정한다.

| | |
|---|---|
| 때로 마추친 책과 사람들에게서..<br><br>주눅이 들게 하는 글을 만납니다..<br><br>주눅이 들게 하는 사람을 만납니다..<br><br><br><br>이 시에 나오는 단어 별 빈도는 다음과 같다.[5] | 때로 1<br>마주친 1<br>책과 1<br>사람들에게서 1<br>주눅이 2<br>들게 2<br>글을 1<br>만납니다 2<br>사람을 1 |

이러한 단어의 빈도분석 프로그램의 로직은 다음과 같이 표현할 수 있다.

```
define WordFrequency as FrequencyTable;

for each document in documentSet {

T = tokenize(document);

for each token in T {

WordFrequency [Word]++;

}

}

display(WordFrequency);
```

위의 분배단계에서 빈도를 저장할 WordFrequency라는 테이블을 지정한 후 각각의 문서 (여기는 하나의 문서)에 대해 분석을 하였다. Tokenize는 문장을 단어별로 분절하는 함수로서 각각의 단어가 발견될 때마다 WordFrequency[Word]를 1씩 증가시켰다.

이제 이러한 작업을 500페이지의 책 1,000권에 대해 수행한다고 하자. 이런 단순한 작업만도 웬만한 컴퓨터로 며칠씩 걸릴 것이다. 심지어 데이터가 수시로 추가/변경되고 또는 더 복잡하고 세련된 분석을 한다면 뭔가 다른 방법을 강구하지 않으면 안 된다. 따라서 위 로직을 프로그램을 여러 대의 컴퓨터에서 동시에 수행시킨 후 결과를 취합하려면 다음과 같이 수정한다.

(분배처리 단계)

```
define WordFrequency as FrequencyTable;
for each document in documentSubset {
T = tokenize(document);
for each Word in T {
WordFrequency[Word]++;
}
}
partition_and_sort(WordFrequency);
sendToSecondPhase(WordFrequency);
```

(결과합산 단계)

```
define totalWordFrequency as FrequencyTable;
```

```
for each WordFrequency received from firstPhase {

FrequencyTableAdd (totalWordFrequency, WordFrequency);

}
```

여기서는 Partition_and_sort()을 통해 앞의 처리결과 WordFrequency[Word] 를 가/나/다/... 순서로 나눈 후 정렬하고 (여기서는 중간합산하는 등의 추가작업은 생략하였다.) 그 결과를 다음 단계에 전달하는 것으로 하였다. 한편 합산단계에서는 이들을 전체 목록 (totalWordFrequency)에 추가하였다.

위 "분배처리 단계"가 MapReduce 에서의 mapping 단계에 해당하고, "결과합산 단계"는 reducing 단계에 해당한다. 결국 mapping 단계는 filter 및 변환의 단계이고 reducing 단계는 종합(합산)의 단계라고 할 수 있다.

한편 mapper와 reducer 그리고 partitioning과 shuffling의 제반 작업이 매끄럽게 이루어지려면 이들 각각이 처리하고 다음 단계로 넘겨주는 데이터가 일정한 구조를 가지는 것이 필요하다. 실제로 이들 mapping 단계와 reducing 단계에 각각 그리고 이들 단계의 연결과정에서 모든 데이터는 (key/value)의 list로 처리된다. 대상 데이터 내용이 어떠하든, mapper, reducer에 지정된 업무로직이 어떠하든 효율적으로 수행되기 위해서 MapReduce에서는 주된 데이터 포맷(data primitive)으로 list와 (key/value) pair를 이용하는 것이다. 다시 말하면 MapReduce에서는 그 어떤 값도 독립해서 존재하지 않으며 모든 값에는 자신에 적용되는 key 가 존재한다. 또한 모든 Key는 관련된 값을 식별해 준다. 다음 그림에 단계별로 전달되는 데이터의 (key/value) pair의 형태와 그 설명과 함께 나타나 있다.

얼핏 보면 매우 복잡해 보이지만 우리는 여기서 기본 규칙을 발견할 수 있다. 즉, 데이터 처리 작업을 map과 reduce의 단계로 나눈 후 각각의 처리 로직을 Hadoop이 지정한 API를 이용하여 프로그램 작성한다. 이때 데이터에 대해서는 포맷을 (kevy/value) pair로 하되 이들이 단계별로 이동할 때 미리 정한 규칙을 따르게 하면 이후의 전반적인 처리는 Hadoop 의 MapReduce에서 관리하고 조정한다. 즉, 개발자가 이들 규칙에 충실히 프로그램을 작성하면 하나의 작업을 수백, 수천 대 컴퓨터에 동시병렬적으로 처리할 수도 있고 결과적으로 전체적인 처리성능은 무한히 커질 수 있는 것이다.

이제 WordCount 프로그램에 대해 이전에 설계했던 작업 로직을 좀 더 MapReduce 규칙에 부합하는 용어로 (여전히 pseudo 코드로) 작성한 것이 다음에 나와 있다.

```
map(String filename, String document) {
for each document in documentSets{
line = readLine(document)
List<String> T = tokenize(document);
for each token in T {
emit ((String)token, (Integer) 1);
}
}
}
reduce(String token, List<Integer> values) {
Integer sum = 0;
for each value in values {
sum = sum + value;
}
emit ((String)token, (Integer) sum);
}
```

map()과 reduce()함수 모두 출력물은 리스트(list) 형태를 가진다는 점만 유념한다면 이 코드는 앞서의 것과 매우 유사함을 알 수 있다. 즉, 이제 분산환경에서 여러 컴퓨터에서 이들 작업을 나누어 할 수 있는 기초가 마련된 것이다. 이제 pseudo 코드가 아닌 실제 구동되는 Hadoop에서의 Java 코드를 보기로 한다.

이제 위의 로직을 구현한 최종적인 프로그램의 모습은 다음과 같다. (단, 이해 편의를 위해 package import, exception, compiler annotation 및 주석 부분을 생략하였다.)

```
public class WordCount extends Configured implements Tool {
public static class WordCountMap
extends Mapper<LongWritable, Text, Text, IntWritable>{
private final static IntWritable one = new IntWritable(1);
private Text word = new Text();
public void map(LongWritable key, Text value, Context context) {
String line = value.toString();
StringTokenizer tokenizer = new StringTokenizer(line);
while(tokenizer.hasMoreTokens()){
word.set(tokenizer.nextToken());
context.write(word, one);
}
}
}
public static class WordCountReducer
extends Reducer<Text, IntWritable, Text, IntWritable>{
public void reduce(Text key,
Iterable<IntWritable> values,
Context context){
int sum = 0;
for(IntWritable value: values){
sum += value.get();
}
```

```
context.write(key, new IntWritable(sum));
}
}
public int run(String[] args) {
Job job = new Job(getConf());
job.setJarByClass(WordCount.class);
job.setJobName("wordcount");
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.setMapperClass(WordCountMap.class);
job.setCombinerClass(WordCountReducer.class);
job.setReducerClass(WordCountReducer.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
FileInputFormat.setInputPaths(job,
new Path(args[0]));
FileOutputFormat.setOutputPath(job,
new Path(args[1]));
boolean success = job.waitForCompletion(true);
return success ? 0: 1;
}

public static void main(String[] args) {
int result = ToolRunner.run(new WordCount(), args);
```

```
System.exit(result);

}

}
```

Hadoop의 모든 MapReduce 프로그래밍은 실제로 위와 같은 모습을 가진다. 따라서 일일이 새로 작성하지 않고 앞에서와 같은 프로그램 template을 변형하여 이용하는 것이 보통이다.

또 다른 예로서 여러 대 자동차에서 측정된 각각의 시간별 속도계 정보가 log 파일에 담겼다고 하자. 아마도 번호판을 key로 하고 관련된 정보가 기록되어 있을 것이다..

AAA-123 65mph, 12:00pm

ZZZ-789 50mph, 12:02pm

AAA-123 40mph, 12:05pm

CCC-456 25mph, 12:15pm

…

Hadoop에서의 mapping 및 reducing 함수는 값(values)만을 받을 수는 없고 언제나 (key, value) 의 pair를 받는다. 이들 각각의 함수의 출력 역시 동일한 원칙이 적용되며 이들 모두 key 와 value 가 다음 단계의 데이터 흐름에서의 list 형태로 산출(emitted)된다.

다만 MapReduce에서는 Mapper과 Reducer의 동작방식이 다른 언어보다는 덜 엄격하다고 할 수 있다. 엄격한 의미의 functional mapping과 reducing에서는 하나의 mapper가 반드시 각각의 입력 항목에 대해 하나의 출력 항목을 산출하고 reducer 역시 하나의 입력 list에 대해 하나의 출력항목만을 만들어 내야한다. 그러나 MapReduce에서는 각각의 단계에서 임의의 수의 (0,1, 또는 심지어 수 백개의) 값들이 산출될 수 있다. reducer 도 마찬가지로 임의의 수의 값을 출력할 수 있다. 단계별 처리과정에서 Key를 기준으로 reduce space가 분할되며 reducing 함수는 많은 값들의 list를 하나의 값으로 변환시킨다.

② MapReduce의 주요 데몬

JobTracker

실제 컴퓨팅 작업을 하는 MapReduce 데몬들도 HDFS와 마찬가지로 master-slave의 구조를 가지며 JobTracker는 master의 역할을, 그리고 TaskTracker는 slave node의 역할을 담당한다.

JobTracker는 응용프로그램과 Hadoop사이의 중간연락을 하는 daemon으로서 파일처리를 위한 실행계획, node할당, task monitoring 등을 담당한다. 즉, 프로그램을 수행시키면 JobTracker가 대상 파일을 찾아낸 후 각각의 node에게 작업할 내용을 할당한 후 이것이 제대로 수행되고 있는지 감독한다. 그리고 이 과정에서 작업이 실패하면 다른 node에게 재 작업을 지시한다. JobTracker는 master node가 이를 수행하는데 Hadoop 클러스터 내에는 단 1개의 JobTracker 데몬이 존재한다.

TaskTracker

(JobTracker가 MapReduce의 master기능을 수행하는 것에 대응하여) TaskTracker는 JobTracker가 지시한 사항을 성실하게 집행한다. 그리고 (HDFS에서의 NameNode와 DataNode와 마찬가지로) 이들 TaskTracker는 JobTracker와 지속적으로 통신하면서 정상적인 동작 유무를 확인할 수 있게 하는데 이상 발생 시 JobTracker는 해당 작업을 클러스터 내의 다른 node에게 중복하여 수행할 것을 지시하게 된다. 특기할 것은 각각의 slave node에는 한 개의 TaskTracker가 존재하지만 이 하나의 TaskTracker는 여러 개의 JVM을 생성시켜서 각각의 slave node 내에서 여러 개의 map task와 reduce task를 병렬적으로 수행시키게 된다.

다음 그림은 1대의 master (NameNode와 JobTracker가 수행됨)와 4대의 slave node (DataNode와 TaskTracker가 수행됨)가 이용되는 모습으로서 백업용의 secondary NameNode가 존재함을 알 수 있다. 한편 이들 master와 slave node들 상호간 통신에서는 SSH 채널이 이용된다.

그동안의 Map/Reduce 전체를 정리하여 도식화 한 그림은 다음과 같다.

n HDFS 클러스터에 파일이 적재됨으로써 MapReduce 입력이 시작된다. 이들 파일은 전체 node에 균등하게 배분되는데 이에 대해 MapReduce 프로그램이 수행되면서 node에서는 mapping task가 시작된다.

n Mapping작업에서 각각의 task는 동등한 것으로서 서로 구별되지 않으며 각 mapper는 그 어떤 입력파일도 처리 가능하다. 각각의 mapper는 자신 컴퓨터 근처에 존재하는 파일을 적재한 후 곧바로 처리에 들어간다.

n mapping 이 끝나면 중간산출물로서의 intermediate (key, value) pair가 각 컴퓨터 사이에서 교환되고 같은 key를 가지는 모든 value들은 하나의 reducer에게 보내진다.

n Reducer에서는 서로 다른 여러 개의 mapper로부터의 산출물이 하나로 병합된다.[6]

끝으로 2004년 Google에서 발표한 MapReduce 논문 (http://research.google.com/archive/mapreduce-osdi04.pdf ) 속의 그림을 소개한다. 앞서의 설명과 그림 안에 기재되어 있는 각 번호의 설명을 통해 그 내용을 이해할 수 있다.

④ MapReduce의 데이터 흐름

Hadoop에서의 MapReduce의 중요성을 감안하여 작업 프로세스가 아닌 데이터의 흐름을 중심으로 정리하여 본다. 아래 그림에서는 각 단계에서의 데이터 흐름을 pipeline으로 보여주고 있다.

Figure 4.5: Hadoop MapReduce의 데이터 흐름에 대한 상세한 모습

입력 파일:

MapReduce task를 위한 데이터는 보통 HDFS상에 존재한다. 파일 포맷은 그때 그때 다른데 line기반의 log 파일, binary 포맷, multiline의 입력 레코드 등 그 어떤 것도 가능

하다. 다만, 용량은 매우 클 것이다.

InputFormat:

입력파일이 분할(split)되는 방식이나 읽어 들이는 방식을 정의하는 class이다.

n 입력으로 사용될 파일 또는 기타의 object를 선택한다.

n 파일을 task로 분해할 InputSplits 을 정의한다.

n 파일을 읽어 들이는 RecordReader 를 생성하는 factory를 제공한다.

Hadoop은 여러 개의 InputFormat 관련 class을 제공한다. FileInputFormat 라는 이름의abstract type이 존재하며; 파일을 대상으로 하는 모든 InputFormat은 이 class로부터 파생된다. Hadoop의 job을 수행하기 시작할 때 FileInputFormat에 읽으려는 파일의 경로를 제시하면 FileInputFormat이 이 디렉토리에 있는 모든 파일을 읽어 들인다. 그런 후 각각의 파일을 여러 개의 InputSplit으로 분해한다. 개발자는 입력파일에 어떤 InputFormat을 적용할지를 JobConf object의 setInputFormat() method를 호출하는 방식으로 정의한다. 표준 InputFormat이 다음 표에 나와 있다.

| InputFormat | 설명 | Key | Value |
|---|---|---|---|
| TextInputFormat | Default 포맷이며 텍스트 파일의 각 line을 읽어들인다. | 각 line의 byte offset | 각 line의 내용 |
| KeyValueInputFormat | 각 line을 key, val pair로 parse한다. | 첫째 tab 문자까지의 모든 내용 | line의 나머지 내용 |
| SequenceFileInputFormat | Hadoop고유의 고성능 바이너리 포맷 | 사용자 정의 | 사용자 정의 |

n TextInputFormat은 입력 파일의 각 line을 별개의 레코드로 취급하지만 별도의 parsing작업은 하지 않는다. 이것은 특히 unformatted 데이터 또는 로그파일과 같은 line기반의 레코드 등에 유용하다.

n KeyValueInputFormat 역시 입력파일의 각각의 line을 별개의 레코드로 취급한다. 다만 TextInputFormat이 line 전체를 하나의 값(value)으로 여기는 반면 KeyValueInputFormat은 각 line을 tab 문자를 기준으로 key와 value로 분해한다는 점이 다르다.

n SequenceFileInputFormat 은 Hadoop에 특수한 바이너리 파일을 읽어 들이는데

Hadoop mapper에 고속으로 읽어 들일 수 있는 몇 가지 부가기능이 제공된다.

InputSplits:

InputSplit은 MapReduce 프로그램에서 map task의 작업단위가 된다. Data set에 적용되는 MapReduce 프로그램은 이를 총체적으로 Job이라고 부르는데 이 job은 수 백 개의 task로 구성된다. Map task는 파일 전체 또는 일부분만도 읽을 수 있다. 디폴트 상태에서 FileInputFormat과 하위 class는 파일을 64 MB 단위의 chunk (HDFS에서의 블록 크기와 동일)로 분할한다. 이 값은 hadoop-site.xml에서 mapred.min.split.size 파라미터를 이용하거나 특정 JobConf object에서 파라미터를 override함으로써 변경할 수 있다. 파일을 chunk 단위로 처리하면 하나의 파일에 대해 여러 개의 map task를 병렬 수행할 수 있어서 성능을 획기적으로 높일 수 있다. 또한 각각의 블록을 한 node에서 다른 node로 옮길 필요 없이 파일을 구성하는 다양한 블록을 클러스터 내의 여러 node에 분배하여 해당 node에서 직접 (locally) 처리할 수 있다. 파일포맷이 chunk단위로 쪼개기 어려울 경우에는 custom의 InputFormat을 작성해서 파일분할의 조건과 방법을 지정할 수도 있다.

RecordReader:

InputSplit을 통해 일의 단위가 지정되지만 그 액세스하는 방법은 정의되지 않는다. 데이터를 source에서 실제로 적재한 후 이를 Mapper가 읽기 수월한 (key, value) pair로 변환하는 일은 RecordReader class가 담당한다. RecordReader instance는 InputFormat에 의해 정의된다. 디폴트의 InputFormat인 TextInputFormat 은 LineRecordReader를 제공하는데 여기서는 입력파일의 각각의 line을 새로운 값(value)로 취급한다. 각 line에 대한 key는 파일에서의 byte offset이다. 입력 시 InputSplit 전체가 완료될 때까지 RecordReader는 계속 호출(invoke)된다. RecordReader가 호출되면 Mapper의 map() method 역시 호출된다.

Mapper

Key와 value가 주어지면 map() method는 (key, value) pair(s)를 Reducer에게 전달한다. 새로운 Mapper instance는 각각의 map task (InputSplit)에 대한 별도의 Java 프로세스 속에서 만들어진다. 이러한 map task는 전체적으로 한의 job input을 구성한다. 각각의 mapper는 다른 mapper와 어떤 통신도 하지 않는다. 이를 통해 각 map task의 신뢰성이 로컬 기기의 신뢰성에 전적으로 좌우된이다. map() method는 key와 value 이외에 2개의 파라미터를 전달받는다.

n OutputCollector object는 collect() 를 통해 (key, value) pair를 job의 reduce 단계로 전달해 준다.

n Reporter object 는 현재 task에 대한 정보를 제공한다. Reporter의 getInputSplit() 는 현재의 InputSplit을 설명하는 object를 반환한다. 또한 map task로 하여금 진행상태에 대한 정보를 시스템 내 다른 요소에게 제공할 수도 있다. setStatus() method 를 통해 사용자에게 상태메시지를 제공할 수도 있다. incrCounter()를 통해서는 shared performance counter를 증가시킬 수도 있다.

Partition & Shuffle:

첫 번째 map task가 종료했지만 각 node들은 여러 개의 다른 map task들을 수행하는 경우에도 map task로부터의 중간산출물을 이를 필요로 하는 reducer에게로 전달하기 시작한다. 이처럼 map의 산출물을 reducer에게로 옮기는 것을 shuffling한다고 한다. 중간단계 key space의 일부가 각각의 reduce node에 할당된다. 이들 subset (이를 "partition"이라고 함)들은 reduce task에게 입력된다. 각 map task는 그 어떤 partition에도 (key, value) pair를 전달할 수 있다. 하나의 key에 대한 모든 값은 항상 그 origin이 어떤 mapper였든 상관없이 병합(reduced together)된다. 따라서 중간산출 데이터의 각 항목을 어디로 보낼지에 대해 map node는 의견 일치를 보아야 한다. Partitioner class는 주어진 (key, value) pair가 어떤 partition으로 갈지를 결정하는데 디폴트의 partitioner는 key에 대한 hash 값을 계산한 후 그 결과에 따라 partition을 할당한다.

Combiner:

Combiner를 이용하면 MapReduce job이 사용하는 대역폭을 절감할 수 있다. Combiner 는 Mapper와 Reducer사이에서 진행되는 것으로서 선택적 적용이 가능한데 이 경우 map task가 수행되는 모든 node에 대해 Combiner class의 instance가 적용된다. 각각의 node에서 Mapper instance가 산출한 데이터를 입력 받고 Combiner가 지정된 작업을 하면 그 결과물을 Reducer에게 보낸다, Combiner는 일종의 "mini-reduce" 프로세스로서 하나의 단위 컴퓨터에서 생성된 데이터만을 대상으로 한다. 이 밖에 Fault Tolerant기능과 Checkpoint 기능이 있다.

앞서의 Word count 프로그램을 예로 들면 각 단어가 발견될 때마다 (word, 1) pair를 산출했는데 예컨대 "주눅이"라는 단어가 2번 발견되면 ("주눅이", 1) pair가 2번 출력되고 따라서 Reducer에게도 2번 전달된다. 그러나, Combiner를 통해 이들이 단위 컴퓨터에서 합산되어서 ("주눅이", 2) pair가 단 한번만 Reducer에게 전달된다. 이처럼 모든 node 에서 여러 번 반복해서 전달되던 항목을 중간합산 하여 각 단어 당 한번씩만 전달됨으로써 shuffle 프로세스에서 요구되는 대역폭을 획기적으로 줄일 수 있게 되고 결과적으로 job 처리속도가 개선된다. 이것 역시 별도의 프로그래밍 작업 없이 driver 프로그램에 다음의 한 줄만 삽입하면 MapReduce 프레임워크가 자동 진행해 준다.

conf.setCombinerClass(Reduce.class);

Combiner step이 MapReduce의 데이터 흐름에 포함되었다.

Sort:

각각의 reduce task는 여러 개의 중간 key에 관련된 value를 합산(reduce)한다. 개별 node에서의 일련의 중간 key는 Hadoop이 이를 자동으로 정렬한 후 Reducer에게 보내진다.

Reduce:

각각의 reduce task에 대해 Reducer instance가 만들어진다. Reducer instance는 사용자가 제공하는 instance로서 job별로 중요한 2번째 단계가 된다. Reducer에게 할당된 partition에서의 각각의 key에 대해 Reducer의 reduce() method 는 단 한번 호출되는데 이를 통해 key에 연결된 모든 value에 대한 iterator와 key를 받는다. iterator에 의해 하나의 key와 관련된 value들이 반환될 때 그 순서는 무작위이다. Reducer는 또한 OutputCollector 와 Reporter object를 파라미터의 형식으로 받게 되는데 이들은 map() method에서와 같은 방식으로 이용된다.

OutputFormat:

OutputCollector에게 제공되는 (key, value) pair는 출력파일에 기록된다. 실제 기록되는 방식은 OutputFormat에 의해 결정된다. OutputFormat 은 앞서의 InputFormat class 와 같은 방식으로 동작한다. Hadoop이 제공하는 OutputFormat의 instance는 로컬디스크 또는 HDFS상의 파일에 기록된다. 이들 모두 일반적인 FileOutputFormat에서 상속된 것이다. 각각의 Reducer는 각각의 파일을 일반적인 출력 디렉토리에 기록한다. 이들 파일은 통상 part-nnnnn 라는 이름을 가진다. (nnnnn 는 reduce task와 관련된 partition id이다.) 출력 디렉토리는 FileOutputFormat.setOutputPath() method에 의해 결정된다. 특별한 OutputFormat을 사용하려는 경우에는 MapReduce job을 정의하는 JobConf object 의 setOutputFormat() method를 통해 지정한다. 제공되는 OutputFormat은 다음과 같다.

| OutputFormat | 설명 |
| --- | --- |
| TextOutputFormat | Default; line을 "key \t value" 형태로 기록한다 |

| | |
|---|---|
| SequenceFileOutputFormat | 뒤에 오는 MapReduce job으로 읽어 들이기에 적당한 형태의 바이너리 파일로 기록한다. |
| NullOutputFormat | 입력을 무시한다 |

Hadoop은 파일에 기록하기 위한 몇 가지 OutputFormat을 제공한다. 디폴트 상태의 instance는 TextOutputFormat으로서 텍스트파일의 각 line에 (key, value) pair를 기록한다. 이를 나중에 MapReduce task를 통해 KeyValueInputFormat class로 다시 읽어들일 수 있는데 이는 사람도 읽을 수도 있다. MapReduce job들 상호간에 이용할 수 있는 더 좋은 중간 포맷이 SequenceFileOutputFormat 인데 이는 임의의 데이터 타입을 파일로 신속하게 serialize해 준다; 이에 대응되는 SequenceFileInputFormat 은 파일을 같은 타입으로 deserialize 하고 앞서의 Reducer가 산출했던 것과 같은 방식으로 다음 Mapper에게 전달한다. NullOutputFormat 은 아무런 출력파일을 만들지 않으며 OutputCollector에 의해 전달받은 (key, value) pair를 무시한다. 이는 reduce() method에서 독자의 출력파일에 기록하고 Hadoop 프레임워크에 의해 추가의 빈 출력파일이 만들지 않으려는 경우 유용하다.

RecordWriter:

InputFormat이 실제로 개별 레코드를 RecordReader 실행을 통해 읽는 것과 마찬가지로 OutputFormat class도 RecordWriter object에 대한 factory역할을 한다. 이들은 OutputFormat에 지정된 대로.개별 레코드를 파일에 기록하는데 이용된다. Reducer에 의해 작성된 출력파일은 HDFS에 남아있으므로 다른 MapReduce job 또는 별도의 프로그램 또는 사용자의 직접개입을 통해 이용할 수 있다.

[1] HDFS에서의 데이터 분배 방식; HDFS는 입력 데이터가 발생하면 즉시 그 파일을 쪼개서 각각의 node에 분배한다. 이때 하나의 파일로부터 만들어진 여러 개 조각을 각각 chunk라고 부르는데 특기할 것은 이들 chunk를 각각의 node에 분배함에 있어 여러 개로 복제하여 분배한다는 점이다. 즉, A라는 이름의 파일을 3조각 내었다면 (이들 각각을 A0, A1, A2라면) 이들 각각의 chunk는 여러 개 (통상 3개 이상)로 복제하여 즉, A0′/A0″/A0‴, A1′/A1″/A1‴, A2′/A2″/A2‴의 여러 복제본을 node에 분배하는 것이다. 그리고 한 걸음 나아가 이들 각각의 복제된 조각에 대해서 이를 분배받은 node는 아무런 차별 없이 동일하게 지시 받은 작업을 수행한다. 따라서 특정 node에 장애가 발생하였을 경우 동일한 내용의 파일조각 (즉, chunk)이 다른 node에 존재할 뿐만 아니라 이미 동시에 수행되고 있는 상태여서 얼마든지 장애node의 내용을 다른 node에서의 동일내용 chunk로 대체할 수 있는 것이다. 물론 이처럼 장애에 대처하기 위해서 전체적인 상황을 감독하는 컴퓨터 (monitoring system)는 데이터를 re-replicate하는 등 끊임없이 관리작업을 수행한다.

[2] 원래 dataset는 특히 행(row)과 열(column) 형식으로 체계화된 데이터를 말하지만 여기서는 (즉, Hadoop에 관한 논의 전체에서) Hadoop의 응용프로그램의 작업대상이 되는 데이터를 뜻한다고 보면 된다.

[3] 단, secondary NameNode의 작업은 실제 HDFS에 대해 이루어지는 것이 아니고 NameNode와의 연락만을 유지한다. 단지, 주기적으로 HDFS의 메타데이터를 유지관리함으로써 유사시 NameNode가 제 역할을 하지 못할 경우를 대비한다. 실제로 이상시에는 관리자가 직접 secondary NameNode를 구동시켜 주어야 한다.

[4] 사용자가 직접 제어하지 않고, 백그라운드에서 돌면서 여러 작업을 하는 **프로그램**을 말한다.

[5] 문장에서 단어를 추출하는 것을 tokenize한다고 하는데 이는 검색을 위한 색인추출(indexing) 의 핵심과정이다. 다만 실제에서는 어근과 어미의 분리, 단수형과 복수형의 처리 및 현재형/과거형/미래형의 처리 등 형태소 분석 문제가 발생할 수 있지만 여기서는 이를 생략하였다. 논의의 주제를 해칠 수 있기 때문이다.

[6] 사실 Hadoop에서 node들 상호간에 통신을 전혀 하지 않는 것은 아니다. 단지 기존 분산시스템에서는 node들 사이의 byte스트림에 대해 프로그래머가 socket을 이용하거나 MPI 버퍼를 통해 명시적으로 marshaling 해야 했던데 반해서 Hadoop에서의 node 간 통신은 묵시적으로 이루어진다는 점에 차이가 있을 뿐이다. 데이터는 key 이름 별로 tagging되어서 Hadoop으로 하여금 관련 정보를 목표 node에 어떻게 전달할 것인가를 알려준다. 다른 한편으로 Hadoop은 내부적으로 모든 데이터 전송 및 클러스터의 topology 문제를 관리한다. Node 들 사이의 통신을 억제함으로써 Hadoop은 보다 안정적이 된다. 개별 node의 장애 발생 시 Hadoop은 다른 기기에서 task를 통째로 재 수행시키는 방식으로 문제를 우회한다. 사용자 차원의 task가 다른 task와 명시적 통신을 하지 않으므로 응용프로그램 상호간 메시지 교환도 없고 checkpoint로 rollback 후 재수행 할 필요도 없다.

출처: <http://www.openwith.net/?p=549>

# [참고]분석기술 개요

2017년 12월 1일 금요일 오전 9:40

## (1) Analytics의 개념?

분석 (Analytics)이란 "감추어진 진실 (hidden insights))을 찾아서 이를 통한 통찰력을 얻고 이를 의사결정에 활용하는 것" 이라고 정의할 수 있다. 즉, 방대한 데이터를 기반으로 계량모델을 수립하고 통계/수리 분석을 실시함으로써 사실(fact)에 근거한 의사결정을 추구하는 것이다. 실제 그 동안 엄청난 데이터가 생산되고 있었지만 제대로 이용하지 못하고 있었는데 컴퓨팅 기술이 발전하면서 분석기법도 새로운 전기를 맞이하고 있는 것이다. 2011년의 IDC 자료에서는 다음과 같이 분석기능을 분류하였다.[1]

## (2) Analytics, BI, 데이터마이닝의 비교

여기서 잠시 Analytics를 기존의 BI 및 데이터마이닝과 비교해 본다. 크게 보면 데이터의 분석 (data analysis)이라는 큰 범주에 속하지만 강조하는 점에 있어 약간의 차이가 있기 때문이다.

| 데이터마이닝 | Predictive Analytics | BI | 기타 | |
|---|---|---|---|---|
| 단서를 찾아 나서는 것 | **문제에 대한 해답을 찾고, 최종적으로 "what next"라는 action으로 나서게 함. – (거래 등) 실제 업무프로세스 자체에 내재화도 가능하다.** | 과거/현재의 데이터를 분석해서 dashboard 및 보고서 작성. 탐색은 역시 analyst-driven 과정 | Descriptive model | Decision model |
| 데이터 속에서 유용한 pattern을 찾는 과정을 자동화하고자 함 | **Analyst-guided 로서 데이터 pattern을 이용해서 복잡한 고객상황의 서술이 가능해지고 나아가 전향적으로 미래예측 시도. – 고객의 행태를 예측** | BI 를 통해 업무담당자는 사업성과 및 trend를 알 수 있다. | 고객을 구매 패턴에 따라 categorize | 특정 action을 취하면 어떻게 될지를 예측 |
| DM은 예측모델을 구축하는 과정 상의 한 단계이다. | **분석가는 DM에서 만들어진 주요변수 들과 pattern들을 가지고 수학적 모델을 작성.** | | | |

| | | | | |
|---|---|---|---|---|
| | **What to do** | 과거/현재, trend | | |
| | **데이터를 통해 개별 고객에게 어떻게 할지 결정하게 함** | Macro한 의미에서의 통찰력 정도 | | |
| | **고객의 lifetime value와 응답 패턴** | | | |
| | **탈퇴(attrition)모델, fraud detection** | | 예: | |
| | **대표적 활동:**<br>**통계/계량 분석**<br><br>**데이터마이닝**<br><br>**다변량 검증**<br><br>**(Multivariate Testing)** | 대표적 활동:<br>보고/KPI/metrics<br><br>자동 Monitoring/<br><br>알림(Alerting와 Thresholds)<br><br>대시보드(Dashboard)<br><br>점수표(Scorecards)<br><br>OLAP (Cubes, Slice & Dice, Drilling)<br><br>Ad hoc query | | |

<표> Analytics, BI, 마이닝의 비교

## (3) 각종의 Analytics 활용 예

빅데이터와 함께 Analytics는 거의 모든 분야에 적용되고 있고 이에 따라 무한히 세분화되고 있는 실정이다. 주요한 활용예를 보면 다음과 같다.

### 마케팅 정보 분석

#### 마케팅 믹스 최적화와 모델링

각종의 판촉효과를 분석해서 최적의 마케팅 활동 배합 (marketing mix)를 찾아내고 매출/수익성을 극대화하기 위해 다변량 회귀분석을 통해 영업 및 마케팅 시계열 데이터를 분석한다. 특히 소비재 제조 및 유통산업에서 특히 많이 이용하며 다음과 같은 형태로 마케팅 투자 및 비용지출을 분석하고 최적화한다.

가격 분석– 소비자의 해당 제품에 대한 가격탄력성 및 이에 따른 가격의 판매실적에 미치는 민감도를 분석.

프로모션 분석 – 프로모션 활동별 (예: 상품진열, 광고홍보, 행사, 할인 등) 매출변화 및 ROI를 분석

### 고객 분석 (Customer Analytics)

마케팅 분석의 한 부분이지만 그 중요성으로 인해 따로 분리하여 생각한다. 특히 고객 행태 및 인구통계학적 분석을 통해 고객행태를 연구하는데 이는 고객의 충성도 (loyalty)와 밀접한 관계를 가진다. 주로 많이 사용되는 기법은 다음과 같다.:

n 고객 segmentation – 모델링기법 (예: 회귀 모델링, 클러스터링, 의사결정 tree)를 통해 전체 고객을 작은 그룹으로 분류.

n 생애 (Life time) 가치분석– 고객이 가진 생애가치 (life time value)를 계량화하고 이를 통해 수익성에 가장 큰 기여를 하는 고객을 분류, 판별한다. 특히 subscription 사업 (예: 통신회사, e-tailing)에 중요.

n 해지 (attrition) 분석 – 계약해지의 위험성이 있는 고객을 식별하고 이를 예방하는 활동을 실시한다.

### 위험 분석

여기서는 신용카드, 금융기관에서의 고 위험 고객을 판별을 위한 분석활동을 살펴본다. 주로 사용하는 기법은 다음과 같다.

n Acquisition 모델링 - 흔히 신청 시점에 획득된 데이터에 대해 실시하고 미래 default위험도 계산.

n 행동평정(Behavioral scoring) - 기존고객의 거래내역 및 신용도 분석을 통해 미래 위험과 수익성을 추정할 수 있다. 또한 risk profile에 기반하여 고객을 분류할 수도 있다.

n Basel II analytics – Basel 위원회가 공표한 Basel II 협정을 통해 금융기관의 위험 관리활동을 개선하도록 되어 있다. 이에 따라 방대한 데이터를 관리해야만 하도록 의무화됨. 이에 대해 적절히 분석하면 risk exposures와 financial/operational risk에 대비하여 적립할 자본금 등을 추정할 수 있다. Base II analytics에는 다음이 포함된다.

n Probability of Default (PD)의 계산

n Loss given Default (LGD)의 계산

n Exposure at Default (EAD)의 계산

n 채권회수 Scorecard 개발

## 웹 분석

다음과 같은 각종 인터넷 데이터를 분석하여 신규고객 창출과 새로운 고객의 확보에 기여하려는 것으로서 크게 다음과 같은 2가지 범주로 나누어 볼 수 있다.

n Off-site 웹 분석 –사이트의 잠재고객을 기회로 보고 사이트를 방문토록 유도하는 것이다. 그 동인 (drivers)과 대화내용이 포함됨. (예: 매출을 독려한 페이지를 추적. 웹 사이트의 상업적 성과를 측정한 후 이를 KPI와 비교하면서 분석함으로써 웹 사이트를 개선하거나 고객반응조사를 통한 마케팅 캠페인에 활용)

n On-site 웹 분석 – 자사의 웹 사이트에 방문한 사람에 대해 측정.

n 웹 사용현황

n 방문자 수

n 사이트에 대한 방문빈도

n 페이지 뷰

n traffic

n 인기도 변화추이

## 인사정보 분석

### 기업

각종의 인사정보 (채용, 보상 및 승진/실적분석. skill 평가, 교육참여 실적 등)를 분석한다. 이를 통해 퇴사위험 직원을 판별하고 사전 조치하며 우수사원에 대해서는 별도 관리가 가능하다. 또한 교육수요의 판별과 조기 대처도 가능하다.

### 스포츠 (Sports Analytics)

최근 기업 뿐만 아니라 스포츠센터 (헬스) 같은 경우에도 효과적으로 활용되고 있는데

스포츠 구단 (AC Milan (축구), Patriots (미식축구) 의 선수관리가 유명하다. 미국에서의 머니볼(Money Ball) 또한 유명한데 이런 사례를 통칭하여 Sports Analytics라고 부른다.

부정방지를 위한 분석 (Fraud Analytics)

은행 (신용카드 분실, 수표위조), 보험회사 (보험사기, 손해 부풀리기, ...)를 비롯한 많은 금융기관에서는 이미 오래 전부터 수 백만 건의 거래를 분석하고 부정거래의 pattern에 해당하는지 여부를 검사하고 있는데 Logistic regression, 신경망, 의사결정 등의 기법이 활용되고 있다.

기타

임상연구 및 임상실험결과에 대한 분석(Clinical Analytics)하기도 하고, 일기예보에 활용하기도 하며 (Weather Analytics) 이밖에 R&D analytics, 공정자료분석 (Process Analytics) 등 거의 전 분야에 적용되고 있다.

## (4) Predictive Analytics

최근 빅데이터와 관련하여서는 특히 예측분석 (predictive analytics)라는 말을 많이 사용한다. 많은 경우 분석그 자체와 혼용되기도 하며 기존 분석론과 큰 차이라기 보다는 강조점의 차이라 생각되지만 워낙 많이 쓰이는 용어이므로 여기서 간단히 살펴본다.

과거 데이터에서 정보를 추출하고 이를 통해 trend와 행동패턴을 예측하는 것으로서 예를 들어 범죄 용의자 식별을 위한 프로파일링 (Profiling), 신용카드 사기의 방지를 위한 (card fraud detection) 활동에 적용된다.

예측분석은 데이터를 단순화함으로써 가치를 높이는 효과가 있다.[2]

이러한 예측분석의 핵심은 데이터를 통해서 설명변수 (explanatory variables)와 예측되는 변수 (predicted variables) 사이의 관계를 파악하고 이를 예측에 이용하는 것이며 흔히 분석모델에 구체적 대상을 적용한 평점작업까지 함께 포함시키기도 한다. 즉,

predictive analytics = predictive modeling+ 모델에 적용하여 산출된 평점 (scoring) + 예측.

말하자면 다음과 같이 구분할 수 있을 것이다.

n 모델은 수학 방정식

n 평점(Score)은 특정 고객 등의 세부 데이터를 적용했을 때 산출되는 값,

## 종류

n 예측모델 (Predictive model) – 소비자의 행태와 함께 과거 실적을 분석해서 미래를 예측하고자 함. 일단 모델이 작성되면 이를 실제 고객과의 거래가 일어날 때 실시간으로 적용할 수도 있다. 최근 avatar analytics라는 말도 나오고 있는데 이는 특정한 환경 조건에서 특정한 고객에게 개별적으로 적용하는 것을 말한다.

n 서술 모델 (Descriptive model) – 데이터에 내재하는 관계를 밝혀내고 이를 계량화하는 것을 말하며 간혹 이를 이용하여 고객 또는 미래의 가능고객을 세분화하고 그룹화하는 것까지 포함하기도 한다. predictive models이 특정한 영역을 중심으로 고객을 분석(예: 신용위험 분석)하는데 비해 서술모델에서는 고객 또는 상품간의 다양한 많은 관계를 식별해 내는 데에 주된 목적을 둔다. (예측모델에서 특정 행위의 유사성에 따라 고객을 rank-order 하는 것과는 달리) 서술모델은 생애주기 (life stage)와 상품선호도에 따라 고객을 categorize한다. 서술모델링 tool을 개별소비자 외에 이들의 집단을 분석하고 예측하는데 이용할 수도 있다.

n 의사결정 모델 (Decision model) – 의사결정 모델에서는 의사결정에 영향 주는 요인들간의 관계를 설명함으로써 의사결정의 과정과 결과를 추측한다. 이를 통해 의사결정의 로직이나 business rules을 개발한다.

## Predictive analytics의 장점

n 고객에 대한 이해를 증진시켜서 미래 고객행위에 대한 통찰력과 예측력을 높인다.

n 추측에 의존했던 것들을 과학에 기반한 합리적 의사결정으로 전환한다.

n risk와 불확실성을 명확히 측정할 수 있게 된다.

n 의사결정에 일관성을 가져오고 이로서 compliance와 고객 서비스를 개선한다.

n 고객관련된 행위의 의사결정을 rule기반으로 함으로써 자동화. 그리고 이러한 결과 ruleset 수립이 단순해지면서 과거 수십 개의 rule이 이제 수백 개로 세분화된다. (물론 자동화는 절차상의 것으로서 실제 의사결정과정은 analyst-guided이다)

n 고객 세분화(segmentation)에서의 정확성을 높이고 이로써 보다 targeted action이 가능해진다.

n 경험적 데이터 분석을 기업의사결정에 활용

## (5) 모델링 (모델개발)

### 모델개발의 단계

모델개발의 단계를 정리하면 다음과 같다.

① 해결해야 할 문제를 정의한다.

② 개발 sample과 관련 실적데이터를 준비한다

③ predictive pattern을 파악하기 위해 데이터를 분석한다.

④ 모델을 개발하고 fine-tune 한 후 검증한다.

⑤ 모델을 deploy하고 실제 현실 적용한 후 평가한다.

이러한 모델 수립에 있어서는 일반적으로 데이터가 많을수록 좋다. 정확성도 높아진다. 그러나 현실적으로 데이터가 한정될 때 다음을 고려할 수 있다.

n Custom 모델 – 보유한 고객에 대한 데이터만을 가지고 특화된 모델 설정

n Pooled 데이터 모델 – 해당 산업 전반과 같은 광범위한 데이터를 가지고 분석한 후 적용이 가능한 분야를 추려낸다.

n Expert 모델 –분석가의 expertise와 주관적인 판단

## 데이터 편향성 (data bias)의 해결

한편 모델 개발의 과정에서는 여러 가지 측면에서 편향성(bias)이 게재되기 쉽다. 여기서 편향성이란 적절하지 않은 샘플의 수 등으로 인한 왜곡 (skewness)을 포함하여 여러 가지가 있는데 분석가가 여러 기법으로 조정을 하게 되며 거절자 추론 (reject inference)이 그 예이다.

### 거절자 추론 (reject inference )

거절된 신청자의 데이터를 이용해서 scorecard의 품질을 높이는 기법이다. 즉, 원칙적으로 샘플은 모집단의 특성을 대표할 수 있어야 하지만 Sample Bias가 생길 수 있기 때문에 이를 최소화시키기 위한 작업이 필요하다. 워낙 자명해서 특별한 의사결정이 필요 없는 집단은 미리 배제할 수 있는데 즉, KGB (Known Goods and Know Bads)는 별도의 과정에서 미리 배제하는 것이다. 일반적으로 모든 모집단은 그 자체에 bias가 있고 따라서 KGB 데이테에 의해 수립된 모델은 그 자체에 결함이 있다.

Reject Inferencing을 통해 이를 교정하는데 즉, 거절되거나, 배제되거나, 충족되지 못한 모집단의 정보를 이용해서 모집단에 대한 우리의 이해를 증진시키는 것이다. 구체적으로 다음과 같은 형태를 가진다.

n Augmentation

n Extrapolation

n Manual Intervention

## 모델의 검증 (validation)

모델의 예측력을 평가하는 기법들

n 발산도 (divergence)

n K-S (Kolmogorov-Smirnov) statistic

n ROC (receiver operating characteristic)

아울러 overfitting을 조심해야 한다. 이는 특정 sample 고객에게 너무 초점이 맞추어져 그에게는 매우 적합하지만 다른 데이터에는 적합치 않은 경우를 말한다.

모델과 관련하여 언급할 것은 단순히 수학이나 통계학의 문제를 푸는 것과는 달리 모델 수립 자체에 많은 탐색과 반복작업(iterative)이 필요하다는 것이다.

어떤 분석 기법을 채택할 것인가?

보통 여러 가지를 적용한 후 결과를 보고 적합한 것을 골라낸다. 또는 availability에 따라 어쩔 수 없이 적용하는 경우도 많다. 일반적으로 다음과 같은 지침을 이용할 수 있다.

n (너무 정밀한 것 대신) 단순한 것, 안정적인 것, 쉽게 설명이 가능한 것을 택하라.

n (너무 시간 오래 걸리는 것이나 세련된 분석보다는) quick-and-dirty 분석을 통해 빨리 결과를 보는 것도 유리하다..

n 즉각 복잡한 모델을 구축하는 것 보다는 상황을 이해하는 데에 더 많은 노력을 기울인다.

[1] Big Data Analytics: Future Architectures, Skills and Roadmaps for the CIO. (2011/9)

[2] Srouce: FICO

이 글은 카테고리: Hadoop기술백서, 빅데이터에 포함되어 있습니다. 고유주소를 북마크하세요.

출처: <http://www.openwith.net/?p=557>

# [참고]분석모델

2017년 12월 1일 금요일 오전 9:40

## (1) 회귀분석

회귀분석을 통해 Pattern 또는 Pattern에 영향을 주는 변수들 간의 관계를 추측, 식별할 수 있다. 예를 들어 특정 마케팅 전략의 유효성을 검증하기 위해 기간별, 매체 별, 마케팅 캠페인의 효과분석을 실시하게 된다.

회귀분석의 장점은 단순 상관관계(correlation)에 비해서 관심 있는 변수(즉, "target" 변수)에 영향 주는 많은 요소를 통제할 수 있다는 것이다. 예를 들어 가격변경 또는 경쟁행위가 특정 브랜드의 매출에 영향 주는 경우 회귀분석 모델을 통해 이들 각 요소가 매출에 끼치는 영향도를 조사할 수 있다.

Regression 분석의 type들

n 선형 회귀분석 – 예측변수 (또는 요소: factor)와 목표변수 (target variable) 사이에 선형관계가 있는 것으로 가정.

n 비선형 회귀분석 – 선형관계가 아닌 경우

n Logistic regressions – target 변수가 이항(二項: binomial)일 때 사용 (1,0 – Accept 또는 Reject)

n 시계열 Regression – 과거의 시계열 데이터로부터 미래 행위를 예측한다.

선형회귀분석

선형 관계. 독립변수를 x, 종속변수를 y라고 할 때 그 관계는 다음과 같다.

y=a+ ßx(1)

단, 실제 연구에서 모집단에서의 a와ß의 값을 추정할 필요가 있으므로

y=a + ßx+ ε(2) (ε는 오차항.)

이 데이터에서 a와 ß 의 값을 추정하는 것. 단, 오차항 ε에 대해 가정이 필요하다. (정규

분포 여부, 기대 값, 분산 등) 이를 감안하여 최소제곱법(least square estimation)에 의한 a와 ß를 추정한다. 여기서 a와 ß의 추정 값의 신뢰도를 검토하기 위해 신뢰구간을 계산하거나 t검정을 이용한다. 독립변수가 여러 개인 경우 F 검정을 이용한다.

## (2) 시계열 (Time series) 모델

여러 기간에 걸쳐 측정된 값들은 일정한 내부 구조 (예: autocorrelation, trend or seasonal variation)을 가지므로 이를 규명하는 것이 유용하다. 이를 위해 trend, 계절 (seasonal) 및 순환적 (cyclical) 요소를 구분(decompose)해 낸다. 주로 많이 활용되는 모델이 autoregressive models (AR) 및 moving average (MA) 모델로서 이에 의거하여 다음 모델이 탄생하였다.

n Box-Jenkins 모델 (1976)에서는 AR와 MA 모델을 결합해서 ARMA (autoregressive moving average) 모델을 탄생시킴. (stationary time series 분석).

n ARIMA(autoregressive integrated moving average models) 은 non-stationary 시계열에 주로 초점이 맞추어짐.

n 그리고 이것이 최근 ARCH (autoregressive conditional heteroskedasticity) 및 GARCH (generalized autoregressive conditional heteroskedasticity) 모델 등으로 발전하였다.

## (3) 의사결정 (Decision) Tree

일련의 의사결정 대안들을 나무 모양으로 늘어놓는 예측모델로서 가장 중요한 것은 각 단계에 맞는 분기점을 찾는 것이다. 즉, 적절한 질문을 찾아내는 것이다. 대부분의 algorithms은 가능한 모든 option을 찾아본 후 여기서 제일 effective option을 골라내는 방식을 취한다. 이를 통해 parent node로부터 2 이상의 child node를 찾아낸다. 이런 방식이 이후의 각 child node에 반복적으로 적용된다. 그런 후 가능한 모든 option을 평가해서 최선의 best split을 찾아낸다. 가장 많이 사용되는 3 가지 방법은 다음과 같다.

n gini,

n Chi-square

n Information gain.

목표변수가 연속적이면 (예: 확률, 소득) F test(=reduction in variance test)를 사용한

다.

Decision Tree의 장점은 다음과 같다.

n 그림이어서 이해하기가 쉽다. 그러다 보니 predictive model 위에 복잡한 이익 및 ROI 모델을 추가할 수도 있다.

n 자동화도 용이하고 SQL로 변환하기도 쉽다. 따라서 다른 정보시스템과 통합도 용이한 편. (predictive modeling과의 결합 이외에도) 데이터 탐색의 단계에서 각종 변수의 영향도를 쉽게 이해할 수 있게 해준다. 실무에서는 수 천 개의 변수와 decision tree가 만들어지고 이를 가다듬는 작업을 한다.

n 특히 변수간 상호작용을 이해하는데 유리하다. 간혹 여러 개의 변수가 합쳐져서 영향을 행사한다. (예: Yamaha motorcycles의 표적시장이 특정 소득계측, 특정 연령대의 남성. 이때 소득계층과 성별을 따로 보아서는 시장세분화를 이룰 수 없다. )

n 누락된 값을 귀속시키는데 사용되기도 한다.

좋은 decision tree가 되기 위해서는 최소한의 변수의 개수는 최소한으로 유지시키는 것이 보기에도 좋고, 해석하기에도 유리하다. 일반적으로 의사결정 tree를 사용하게 되는 경우는 다음과 같다.

n 문제의 성격이 binary일 때 (0/1, yes/no).

n 결과의 해석이 필요할 때

n 시간이 촉박할 때

n 프로젝트 초기의 initial data exploration 시에 사용

사례: 대출상환 연체 분석

최근 2~5년 사이에 연체율이 높아져서 여신정책을 엄격히 운영하려 한다.

은행 측에서는 연체가 발생하는 대출자의 특성(profile)을 파악함으로써 정상적으로 상환하는 고객과 어떤 차이가 있는지를 밝히는 한편 신규 대출시 연체집단 성격에 부합하는 사람에게는 대출을 자체하고자 하였다.

이 의사결정 tree는 10,000명의 대출자를 대상으로 작성되었다. 이들 중 200명이 연체

를 하여서 연체발생율은 0.2%였다. 이러한 의사결정 tree를 이용해서 상환연체 집단의 성격을 밝히고자 하였다.

첫 단계에서 한쪽 집단(A)은 연체율이 0.4%인 반면 다른 집단(B)은 75%에 달하였다. 조사결과 A 집단의 경우 월 평균소득이 월 불입액의 1.2배 이상인 반면 B집단은 그 이하의 소득을 가지고 있었다.

따라서 B 집단의 특성을 가지는 사람에게는 대출을 엄격히 제한해야 한다는 결론에 도달하였다.

나아가 대출신청자의 특성에 따라 그 집단에 해당되는 질문을 던짐으로써 더욱 세분화시켜 나갈 수 있었고 최종적으로 연체율이 50%가 넘는 3개의 소 집단을 찾아낼 수 있었다.

여기에 그러한 의사결정 tree의 분석 예가 나와 있다.

## (4) 클러스터링 (clustering)[1]

비슷한 특징을 가지는 소집단으로 묶어내는 것을 말하며 단방향(an undirected)의 데이터 마이닝 기법으로 사용되기도 한다. 즉, 특정한 가설을 세우지 않고도 숨겨진 pattern을 찾는 것으로서 target 변수가 없다.

Clustering 기법의 대표적인 것으로 다음 2가지가 있다.

n 비 계층적 (non-hierarchical) 기법 – N개의 구성인자를 M개의 클러스터로 분류한다. K-평균 알고리즘(K-means algorithm)이 대표적인데 여기서는 주어진 데이터를 k개의 클러스터로 묶으면서 각 클러스터와 거리 차이의 분산을 최소화하는 방식으로 동작한다.

n 계층적 (hierarchical) 기법- 클러스터링 하는 과정에서 여러 개의 내포된 클러스터(nested clusters)가 만들어진다.

세부적으로는 다음과 같은 분석기법을 동원하여 클러스터링을 하게 된다.

n Partitioning 기법

n Density-based 기법

n Grid-based 기법

n 모델-based 기법

n Outlier 분석

n 기타

클러스터링은 주로 다음과 같은 경우에 많이 활용된다.

n 주로 세분화 (segmentation) 작업에 많이 사용된다. 특히 고객 세분화 (customer segmentation), 용도, 크기, brand, 맛 등에 따른 상품배합 세분화 등이 대표적이다.

n anomaly 탐지 (예: fraud transaction의 식별). 이는 정상적 거래( "normal" cluster)와 비정상 거래를 구분하거나 약품 부작용의 검사, 통신사에서의 calling 패턴 분석을 통한 fraud 탐지에 사용된다.

n 대규모 데이터를 나누어서 작은 데이터 그룹으로 분리. (예: logistic 회귀분석의 결과를 보다 작은 클러스터 집단으로 분류해서 다른 행동양식을 비교할 수 있다.)

사례: 유통점 고객분류

채소 판매업자가 단골회원카드를 150만명의 고객에게 발급하면서 구매행태에 따라 5개의 그룹으로 분류하였다. 그리고 이들 각각의 그룹에 대해 특화된 마케팅 활동을 전개하였다.

"신선식품 선호그룹"으로 명명된 한 고객집단은 유기농과 신선채소 등을 주로 구입하는 행태를 보였는데 이들에게 신선함과 1년 내내 이용할 수 있는 유기농을 홍보하였다.

"편의성 추구 그룹"으로 명명된 또 다른 그룹은 미리 조리된 식품이나 조리가 간편한 제품을 선호했으므로 냉동식품 등을 홍보하였다.

이런 방식으로 특화된 제품과 함께 특별히 준비된 메시지를 집중 전달함으로써 마케팅

활동의 효과를 배가시킬 수 있었다.

위의 예에서 채소 소매상은 특별히 신선식품 선호집단을 처음부터 분류해 낼 필요가 없었다. 대신 지속적으로 유사성 (예: 유사한 소비자 행동)을 찾아내는 작업이 필요했다. 즉, 설명 또는 해석이나 왜 라는 질문 없이도 pattern 발견이 가능하다. 다만 그 결과 상정되는 클러스터 (resulting cluster)는 그 자체로는 무의미하며 광범위하게 프로파일되어서 자신의 특색(identity)을 확인할 수 있어야 의미가 있다.

교차분석 (Cross Tabulation)

변수 사이의 관계를 검토하면서 하나의 변수가 다른 변수와 관련이 있는지, 있다면 그 관계에 다른 변수를 도입하여 이들 변수가 최초의 관계에 영향을 미치는지 여부를 보게 된다.

2개의 변수 A, B를 각각 여러 카테고리(A1, A2, ···, Am) (B1, B2, ···, Bn)로 분류한 후 (Ai, Bj)의 빈도표를 작성하면 집계표가 작성된다.

## (5) 연관성 규칙

연관성 규칙을 중심으로 2개 그룹간의 연관정도 (degree of association)를 분석하는 것이다. 연관성 규칙에는 유사그룹의 분류 (Affinity grouping)와 규칙귀납 (rule induction)이 대표적인 기법이며 대규모 데이터로부터 특정 군집에 반복 출현할 가능성에 대한 규칙을 만들어서 pattern을 식별해서 내는 것이다.

대표적인 예가 상점에서 묶음 상품을 만들어 내는 것, 웹 상에서 추천 시스템 구축 등이다.

다만 이러한 연관성 규칙이 유용하기는 하지만 자칫 수 많은 상품군에서 무한히 많은 조합을 만들어 내고 당연한 사실에 대한 중복정보를 생산할 수 있기 때문에 측정기준을 만들어야 한다. 그러나 심도 있게 사용하면 유용하다. (예: 시간대별 구매행태의 변화양상 등.)

또한 일반적인 의사결정 tree는 모든 상황을 대상으로 하고 (exhaustive) 또한 각각 변수성을 전제로 하지만 (mutually exclusive) Induction rules은 그렇지가 않다. 연관성 규칙의 예는 다음과 같다.

n 우유와 바나나를 사는 사람은 시리얼 (cornflake)을 살 확률이 높다.

n 다이어트 식품을 사면 다른 제품을 살 때도 다이어트 제품을 우선적으로 산다.

n 맥주 사면 맥주 안주도 함께 산다.

## (6) 신경망

신경망은 일종의 random prediction을 하고 이에 대해 데이터 각각을 적용시키면서 체계적으로 업데이트하는 방식으로 학습시킨다. 클러스터링 기법 등을 중심으로 한 자율학습 (unsupervised learning). 흔히 fraud detection, 신용평가 (credit scoring), 상권분석 (store clustering) 등에 많이 이용된다.

신경망 (또는 인공신경망)은 인간의 두뇌활동을 본 따서 pattern의 탐지, 경험에 기반한 학습과 예측을 한다. 그러나 아직은 인간과 같은 직관력이 없어서 한번에 하나씩만 (one-record-at-a-time) 학습한다는 면에서 다른 기법과 크게 다르지 않다. 다만, 앞으로 큰 발전이 기대된다.

신경망으로부터 얻어진 결과는 설명이나 해석이 쉽지 않아서 그 과정을 black-box로 처리하는 경우가 적지 않다. 즉, 해답은 있는데 그 도출과정을 정확히 추적하기 어려울 수도 있다는 뜻이다. 따라서 입력변수 및 예상변수에 대해 어느 정도 그 내용과 성격을 잘 이해하고 있을 때 또는 모델링 하고자 하는 것이 무엇인지 그 업무에 대해 잘 알고 있을 때 사용하는 것이 좋다. 물론 예측 이외에도 clustering, outlier 탐지 및 변수 선택 등에도 사용될 수 있다.

데이터를 준비(Data preparation)하는 데에 있어 복잡한 과정이 게재되는데 특히 데이터의 표준화가 중요하다. 그리고 민감도 분석을 통해 입력변수의 상대적 중요성을 이해할 수 있으며 데이터 상호간의 관계가 매우 복잡할 때는 그 관계를 체계적으로 분석하는 것이 중요하다. 예컨대 이익증가를 위한 제반 요소들의 관계를 신속, 정확하게 밝혀 내고자 한다면 거래의 빈도가 높고 비정상적 (abnormal) 데이터 패턴이 있는 경우 – 예컨대 fraud –신경망 모델을 사용하는 것이 좋다. 이때 신경망 모델은 일종의 블랙박스로 동작하며 여러 입력변수를 선택해서 이들 관계를 찾아내는 등의 수학적 모델의 정립이 중요하다. 이러한 신경망 기법은 이미 많은 상용 및 오프소스 S/W에 모듈형태로 구현되어 비교적 손쉽게 적용할 수 있는 상태가 되었다.

## (7) 분류 (classification)

이미 알려진 몇 개 그룹에 속하는 다변량 관측치로부터 각각의 그룹이 어떤 특징을 가지고 있는지 분류 모델을 만든 후, 새 관측치가 어떤 그룹에 분류될지를 결정하는 것으로서 분류기법의 대표적 예는 다음과 같다.

n 판별분석 (discriminant analysis)

n 선형 판별 함수

n 비선형 (2차) 판별함수

n 의사결정 tree induction

n CART(classification and regression tree)

n C5.0 알고리즘

n 회귀분석

n Logistic 회귀분석

n 베이지언 (Bayesian) 분류

n Naïve Bayesian – Bayes의 조건부 확률론에서 출발한 것으로서 예측변수가 독립적이라고 가정.

n SVM (Scalable Vector Machine) – 데이터가 가진 다양한 패턴을 클러스터링, 분류 ranking 등을 이용해서 탐색하고 발견하는 것.

n K-NN(k-nearest neighbor) -

n 규칙 기반의 분류와 사례기반추론(case based reasoning)

## (8) 기계학습

기계학습 (Machine learning)은 인공지능의 한 종류소서 데이터로부터 스스로 학습하는 것을 말한다. 예컨대 이메일에 대해 꾸준히 학습하면서 spam과 non-spam 폴더로

자동 분류해주는 것을 들 수 있다.

기계학습의 핵심은 다음의 2가지 이다.

n 데이터의 표현 (Representation of data instances)과 이들에 대한 평가를 위한 함수

n 일반화 (Generalization)란 현재의 모형이 새로운 데이터에도 그대로 적용될 수 있도록 하는 것을 말한다.

기계학습의 종류는 다음과 같다.

n 지도학습 (Supervised learning)

n 자율학습 (Unsupervised learning) 모델:

n 강화학습 (Reinforcement learning)

n 학습방식 학습 (Learning to learn)

n K 근접 이웃(k-nearest neighbors algorithm (k-NN)) – 관찰치 특성을 기준으로 훈련 샘플 중에서 가장 가까운 관찰치들을 분류하는 방법.

n 공간정보 예측모델 )Geospatial predictive modeling)

여기서 특히 K-NN 알고리즘은 기계 학습의 방법 중에 가장 간단한 방법 중 하나로 분류되고 있다.

기존의 'nearest neighbors(NN)' 기법은 새로운 instance를 분류하는 문제에서 가장 유사한 instance를 찾아서 그것과 같은 class에 새로운 instance를 일방적으로 분류했기 때문에 noisy한 dataset에 대해서는 좋지 않은 성능을 나타내었다. 이에 대한 보완책으로써 나온 것이 'k-NN'이며, NN이 가장 근접한 1개의 데이터만 보는 것과는 달리, 가장 근접한 k개의 데이터에 대한 majority voting, 혹은 weighted sum의 방식으로 classification을 하게 된다. 즉, NN은 1-NN이라고도 볼 수 있겠다.

다만 단점으로는 모수(parameter)가 적용되는 대부분의 모델들이 그렇듯, k-NN 역시 모수 k를 정하는 것이 쉽지 않다. 일반적으로는 여러 개의 k값에 대해 modeling을 한 후, validation set에서 가장 성능이 좋은 model의 k값을 선택하게 된다. (노이즈가 심한 데이터일수록 k값이 큰 것으로 알려져 있음)

한편 공간정보 예측모델에서는 특정 event의 발생은 일정하지도 않고 그렇다고 무작위적인 것도 아니라고 본다. 즉, 공간상의 요인 (spatial environment factors) 즉, 사회문화적, topographic, 등의 요인들이 영향을 준다는 것이다. Geospatial predictive 모델에서는 geographic filter를 통해 이들 영향 주는 것들과 제약사항을 모델로 표현하고 이를 예측에 활용하고자 한다.

[1] 클러스터링, 집단화, 다발짓기, 군집화, 군집표집(郡集標集) 등으로 표현하기도 한다.

출처: <http://www.openwith.net/?p=559>

# [참고]Analytics 도구

2017년 12월 1일 금요일 오전 9:41

## (1) 개요

원본 데이터(raw data)로부터 의미있는 통찰력을 찾아내는 분석(analytcis)을 위해서는 수학, 통계학 등의 엄밀함을 갖춘 모델링 작업이 필요하지만 그렇다고 맨 바닥부터 작업하는 것은 효율적이지 못하므로 다양한 도구를 사용하게 되며 여기서 주요한 것들을 살펴본다.

주의할 점은 여기 열거된 것 이외에도 각각의 상황에 맞는 수 많은 분석도구들이 있으며 이들 모두 나름의 장단점을 가진다는 점이다.

도구 못지 않게 도구를 다루는 사용자의 분석력과 업무지식이 중요한 이유이다.

한편 개별 분석도구를 살펴보기에 앞서 PMML (Predictive Model Markup Language)라는 것이 있음을 유의하자. PMML은 XML 기반의 markup 언어. predictive analytics 및 data mining관련 모델을 정의하는 언어로서 관련 분야의 소프트웨어 들은 PMML-compliant applications을 공유할 수 있다. 즉, 한마디로 말하여 특정 공급업자에게 매이지 않는 (vendor-independent) 모델 정의방식이라 할 수 있다.

## (2) 상용

n MS Excel:

n SAS:

n SPSS Modeler: 원래는 SPSS Clementine이었으나 IBM 인수.

n Statistica: StatSoft사의 개발품. 통계, 데이터 분석, 데이터마이닝, data visualization 가능.

n Salford Systems: 다양한 예측분석 및 데이터마이닝 지원. 특히 classification 및 regression tree algorithms에 강점이 있다.

n KXEN: 분석 자동화 도구로 유명하다.

n Angoss: 쉽고 이해하기가 좋다.

n MATLAB: MathWorks 개발. matrix 계산, 함수 및 데이터의 plotting 등에 강점. 다양한 add-on toolboxes를 통해 확장가능. Matlab은 상용제품이지만 많은 clone제품이 있음.

n Mathematica

n Stata

## (3) 오픈소스

n R

n WEKA: Weka (Waikato Environment for Knowledge Analysis).

n Python의 다양한 라이브러리 활용

**R**

R 은 원래 1970년대의 S라는 통계프로그램에 기반하여 개발되었다. S는 AT&T에서 개발되어 Insightful Corporation라는 회사에 라이선스 제공되었는데 이를 참조하여 뉴질랜드의 Ihaka 등이 1990년대에 R을 개발하였다. GPL 라이선스 조건을 가지는 공개 소프트웨어로서 다음과 같은 영역에서 장점을 가진다.

n 데이터 처리, 문자 및 수치 계산 등이 효율적이다.

n 행렬식 등의 처리와.해쉬(hash) 테이블, 정규식 (regular expression) 표현 등 기능이 풍부하다.

n 데이터 분석, 통계 함수 등이 풍부하다.

n 개발 초기부터 그래픽 기능이 뛰어나다.

n 프로그래밍 언어로 동작이 가능하다.

단, 올바른 사용을 위해 R에 대해 다음을 유의할 필요가 있다.

n R은 데이터베이스는 아니다. 단, DBMS에 연결은 가능하다.

n 원래 명령어 방식이다. 최근 일부 GUI 가 제시되고 있다. (R Studio와 Java, Tcl/Tk 등)

n 모든 계산이 메인메모리에서 진행되므로 메모리 관리에 유의할 필요가 있다.

n 스프레드 쉬트로 사용할 수는 없으나 Excel 등에 연계는 가능하다.

무엇보다도 중요한 것은 CRAN (Comprehensive R Archive Network)에서 패키지를 고를 수 있고 이를 통해 기능의 확장이 가능하다는 점이다. 즉, 다양한 라이브러리와 소스코드를 이용할 수 있다.

R은 배우기에 조금 까다로울 수 있으며 흔히 간단한 것은 매우 쉽고, 어려운 것은 매우 어렵다고 이야기한다.

절차적 프로그래밍 언어로서 일반적으로 명령어 방식을 이용하지만 R Studio가 이용가능하고 Eclipse에 plugin방식으로[1] 이용할 수도 있다.

## (4) 빅데이터 분석 관련 프로젝트

Storm & Kafka

Storm과 Kafka는 스트림 (stream) 처리기술로서 이미 많은 기업에서 활용하고 있다. Storm은 Twitter에서 개발한 "분산형 실시간 연산시스템"으로서 Hadoop에서 batch처리를 했던데 반해 이를 실시간 처리하게 해준다.

Kafka는 for its part is a messaging system developed at LinkedIn에서 개발한 메시징 시스템으로서 활동 스트림 (activity stream)과 그 배후의데이터 처리 파이프라인 (data processing pipeline)의 기반기술로 작용한다.

이들 2가지를 이용해서 stream작업을 하면 실시간 처리를 하면서도 성능은 선형에 가깝게 개선할 수 있게 된다. 즉, Storm과 Kafka를 이용해서 실시간으로 초 당 수천~수만 건의 메시지를 처리할 수 있다는 점에서 기존의 ETL과 데이터 통합에 새로운 지평을 열고 있다.

Storm과 Kafka 같은 Stream 처리 솔루션은 또한 메인메모리 기반으로 (in-memory) 실시간 의사결정 지원시스템으로서의 분석작업에 유용하다. Hadoop에서의 배치처리를 극복하기 때문이다.

## Drill & Dremel

Drill과 Dremel 은 대규모, ad-hoc query를 하면서도 시간지체가 거의 없어서 (radically lower latencies) 특히 데이터의 탐색 (data exploration)에 유용하다. 초당 petabyte 규모의 데이터를 스캔할 수도 있어서 ad hoc query는 물론 시각화 (visualization) 작업에도 유용하다.

Drill과 Dremel은 데이터 관련 기술업무 이외에 현업분석자에게도 매우 유용하다.

Drill은 Google의 Dremel 에 해당하는 오픈소스 솔루션이다. (Google은 이미 BigQuery 라는 이름으로 Dremel-as-a-Service를 제공하고 있다).

Drill과 Dremel 모두 Hadoop에 친화적이어서 MapReduce와의 연계가 손쉬우며 Sawzall, Pig 그리고 Hive,에 이르기까지 Hadoop위에서 동작할 수 있도록 많은 인터페이스가 개발되어 있다.

출처: <http://www.openwith.net/?p=561>

# [참고]분석 사례

2017년 12월 1일 금요일 오전 9:41

## (1) 분석 CRM (Analytical customer relationship management)

분석 CRM 이야 말로 Predictive Analysis이 가장 활발하게 이용되는 분야이다. 전사적, 부서 차원에 일관성 있게 적용할 수 있는 통일적 관점을 제공한다. (holistic view of the customer) 이를 통해 전 부서가 고객의 단계별 상황에 맞춘 대 고객행위를 할 수 있게 된다. 이하에서의 여러 가지가 이러한 분석 CRM의 한 부분이다.

### 채권회수 (Collection)

채권회수를 위해 예컨대 회수비용을 최소화하면서도 어떻게 내부자원과 외부자원을 최적으로 할당할지를 알려준다. (예: 최적의 회수기관, 회수전략, 법적 조처, 기타의 활동)

### 교차판매 (Cross-sell)

고객별 지출규모, 소비행태를 분석해서 교차판매 (묶음판매), 추가판매 등을 유도한다.

### 고객 유지 (retention)

경쟁이 심한 경영환경에서 고객만족을 높여서 경쟁사에게 고객을 빼앗기는 것을 미리 막아야 한다. 흔히 고객이 이탈한 후에야 사후적으로 조처를 취하는 경우가 많지만 무엇보다 중요한 것은 이탈조짐을 미리 파악하고 대비하는 것이다. 여러 행동 유형을 분석해서 이탈의 위험이 있는 고객에게 불만의 원인이 무엇인지를 파악하고 조금 과도할 정도의 보상을 제공하는 것도 방법이 될 수 있다. 최악의 경우는 기업이 알아차리지 못하는 사이에 고객이 아무런 말도 없이 떠나버리는 것이다. Predictive analytics를 통해 이러한 행태를 미리 예측하고 적절한 조처를 취하는 것이 중요하다.

### Direct marketing

고객을 식별해는 것과는 별개의 것으로 predictive analytics를 이용해서 최적의 상품배합, 마케팅 자료, 커뮤니케이션 채널 및 timing 등을 선택할 수 있다. 이를 통해 cost per order or cost per action을 낮추는 것이 중요하다.

## (2) DM 기법을 이용한 Predictive Analytics의 실제 예

Direct Marketing 활동을 함에 있어 고객의 구매행태 등 제 요인에 의해 분류한 후 각각의 소 그룹에 맞는 마케팅 전술을 적용하고자 한다. 이를 위해 다음의 단계별 접근을 취하였다.

① 예측변수를 이용해서 고객을 등급 메긴다. (Rank Your Customers)

Predictive analytics의 중심에는 예측변수 (predictor)라는 것이 있는데 이는 각각의 고객에 대해 측정된 값을 의미한다. 예컨대 recency[1] (최근 몇 주 전에 마지막으로 구입했는가)를 통해 새로 마케팅 캠페인을 했을 때 호응도를 가늠해 볼 수 있다. 가장 최근 구입한 사람이 가장 호응도가 높을 것으로 예상된다는 점이다

이외에도 목적에 따라 다양한 예측변수 (predictors)를 선택할 수 있다. 온라인 상점의 경우 역으로 상거래 사이트에 가장 짧은 시간 머무른 사람은 탈퇴의 가능성이 높다고 볼 수 있다. 이 경우, 고객유지 캠페인을 이들에게 집중할 필요가 있다.

② 여러 가지의 예측변수를 가중치 적용하여 조합한다.

효과를 극대화하려면 위의 예측변수를 여러 개 선택해서 조합하는 것이 좋다. 그리고 이를 공식화(formula) 하면 예측모델이 된다. 위의 예에서 recency 에 추가하여 개인소득을 추가시키고 다음과 같이 그 가중치를 반영한 공식을 만들 수 있다.

2 x recency + personal income

이제 모델이 만들어졌다. 위의 경우에 우리는 단순한 선형관계를 가정하였으나 이 외에도 다음과 같이 조건에 따른 규칙을 정의함으로써 business rule을 정의할 수도 있다.

If 고객이 지방거주자이고, and 월 지출액이 높다면,

then 계약관계를 지속할 확률이 높다.

또는

If 고객이 도시거주자이고, and 신 상품 구매비율이 높다면,

then 이번 캠페인에는 참여하지 않을 것이다.

실제에서는 아마도 이러한 모델이 훨씬 세분화되고 복잡할 것이다.

③. 실제 고객 데이터를 대상으로 컴퓨터를 이용하여 모델을 수립한다

실제 가장 중요하고도 어려운 점은 예측모델을 만드는 것이다. 실제로 현장에서의 복잡한 상황과 함께 이론적으로도 세분화된 모델이 매우 많기 때문에 이들을 엄선하고 수정해서 자신의 상황에 맞는 모델을 수립하는 것이 중요하다.

각종 현장상황이 반영된 모델을 컴퓨터 상에 소프트웨어로 구축한다. 실제로는 많은 시행착오가 있을 것이다.

Wisdom Gained: A Predictive Model is Built from Customer Data

④ 적용결과를 다각도로 검증한다.

컴퓨터상의 모델링까지 완료되면 이를 검증할 필요가 있다. 이익곡선(profit curve) 상에서와 실제 현장에서의 캠페인을 통한 이익증가 추이를 비교해 본다. 물론 이러한 이익증가는 고객을 얼마나 정밀하게 분류 (ranking)해서 세분화된 캠페인을 실시했는가 하는 것이 중요할 것이다.

처음 얼마 동안은 고객접촉이 늘어날수록 이익도 증가한다. 하지만 초기 얼마를 지나면 이익증가가 감소하고 약 70% 고객접촉 이후부터는 오히려 이익이 감소됨을 보여준다. 실제 시간과 예산을 투입해도 캠페인에 따른 비용증가에 비해 매출증가가 이루어지지 않기 때문이다.

전형적인 이익곡선의 모습

위 그림에서는 처음 얼마 동안은 고객접촉이 늘어날수록 이익도 증가한다. 하지만 초기 얼마를 지나면 이익증가가 감소하고 약 70% 고객접촉 이후부터는 오히려 이익이 감소됨을 보여준다. 실제 시간과 예산을 투입해도 캠페인에 따른 비용증가에 비해 매출증가가 이루어지지 않기 때문이다.

결론

물론 이러한 과정은 데이터의 축적, 모델 선정과 이에 대한 평가 등 각 단계마다 컴퓨터의 도움을 받을 수 있다.  그러나 이 못지 않게 사람 (분석가와 업무담당자)의 노력과 헌

신이 중요하다.

## (3) Fraud Detection

어디서나 Fraud 가 발생한다. 다양한 종류가 있다. 부정확한 대출신청, 사기성 거래 (both offline and online), 명의도용 및 부정 보험청구. 신용카드 부정사용. 등등. 여기서 predictive model 을 통해 잘못된 거래를 방지하거나 회피하고 이에 대한 노출위험 자체를 낮출 수 있다.

Predictive modeling에는

n risk-scoring 을 통해 특별히 감사해야 할 유형을 식별. (예: 가맹점 (franchisee) 영업에서 지역분석을 위한 10개의 예측변수(predictors)를 개발하고 이들에 대해 가중치 부여. 이러한 접근법은 부정수표의 사용, 기타 여러 사례에 이용이 가능하다)

n 미국 국세청 (IRS) 에서도 탈세방지에 적용.

n web fraud detection

다음은 특히 전자금융거래에 대한 탐지에 대한 한 보고서이다.[2]

[1] 고객의 가치측정에 사용되는 측정기법에는 크게 2가지가 있다. 우선 전통적 RFM 분석에서는 Recency (얼마나 최근 구입했는가), Frequency (얼마나 빈번하게 구입했는가), Monetary amount (총 구입금액이 얼마인가)의 3가지 정보를 통해 제품구입가능성이 높은 고객들을 추려내다. RFM기법이 사용에 편리한 점은 있지만 개별 고객의 수익기여도를 직접적으로 측정하지는 못한다. 이러한 RFM의 한계를 극복한 것이 고객 수익성 분석으로서 여기에는 고객 평생가치 (customer lifetime value) 평가법과 고객실적 평가법이 있다.

[2] 이상 금융 거래 탐지 및 대응 프레임워크, TTA (한국정보통신기술협회) 정보통신

단체표준 (TTAK.KO-12.0178), 2011년 12월 21일

출처: <http://www.openwith.net/?p=563>

# [참고]빅데이터 적용 방법론

2017년 12월 1일 금요일 오전 9:42

## 1. 개요

빅데이터 프로젝트 역시 큰 그림에서는 일반적인 개발 프로젝트와 많은 공통점을 가진다. 다만, 데이터 처리를 위한 새로운 프레임워크의 적용이라는 측면과 빅데이터를 대상으로 하는 만큼 비정형 (unstructured 및 semistructured)의 대량 데이터에 대한 분석이라는 면에서 몇 가지 과거와 다른 요소가 개입되는 것이다. 여러 가지 이론제시가 되고 있으나 여기서는 IDC의 보고자료와 Sprenger의 성숙모델을 소개한다.

## (1) IDB의 빅데이터 분석 성숙모델

다음은 IDC의 초기 자료에 제시된 모습[1]은 다음 그림과 같다.

빅데이터 성숙모델

이미 선진 기업에서는 객관적 데이터 분석을 중심으로 한 합리적 의사결정을 하는 조직과 그렇지 못하고 직관에 의해 움직이는 조직 사이에는 커다란 차이가 불가피한 것으로 내다보고 있다.

최근 (2013/3월) IDC는 "Big Data and analytics (BDA) maturity model"라는 보고서를 발간하였다.[2] 여기서 단계를 나누고 각각의 특징을 나열하면서 요구되는 대응방안을 밝히고 있다. 여기서는 조직으로 하여금 다음 사항을 추진하도록 제반 항목을 정리하였다.

n 데이터분석능력과 성숙도 (BDA competency & maturity)를 분석

n 단기 및 중장기 목표를 향해 나아감에 있어 개선방안 수립을 위한 기준점 (baseline)을 제시

n BDA 관련 기술, 전문인력 충원 및 기타 투자의사결정의 순위를 정함

n 현업부서간 또는 현업부서와 IT부서 간의 성숙도 차이 (maturity gaps)를 드러내고 해소하는 것.

이를 위해 현재 상태와 지향하는 목표 사이의 gap을 측정하고 BDA maturity 와 competence를 높이기 위한 방안을 제시한다. 5개의 stages와 5개의 핵심조처항목, 성과목표, 행당방안을 제시하고 있다.

IDC의 BDA Maturity Model은 발전단계에 따라 다음의 5가지로 분류하였다.

Stage 1 — Ad Hoc: 조직이 하나의 사업상 이슈에 대해 실험적으로 분석에 착수한다. 이때는 BDA를 위한 전략도 없고 경영층으로부터 특별한 지원을 받지도 못하는 상태이다. 단지 기존의 기술에 의존하면서 비용절감을 위해 오픈소스 또는 클라우드 기술을 조금 활용할 뿐이다.

Stage 2 — Opportunistic: 조직이 앞서의 파일롯 프로젝트의 수행경험으로부터 교훈을 얻고 이를 새로운 사업상 요구사항에 적용한다. 이제 새로운 프로젝트에 대해서는 별도의 예산지원을 받게 되며 중간 관리자의 지원도 받을 수 있게 된다. 그러나 아직도 전사적 지원은 받지 못하는 상태이며 성과에 대한 객관적 측정기준도 가지지 못한다.

Stage 3—Repeatable: 이제 조직은 예산지원이 되는 중소규모의 BDA 프로젝트를 반복적으로 실행하며 사업본부 단위 정도에서 지원과 지휘를 받게 된다. 즉, 데이터의 수집, 감도, 통합 프로세스가 사업본부 단위에서는 발생한다. 그러나 전사적인 data governance나 보안정책 상의 지침을 받지는 못한다. 내부직원에 더하여 외부의 전문가의 서비스도 받는 정도가 된다.

Stage 4 —Managed: 조직 자체에서 BDA 프로그램의 표준이 제정되며 여러 사업본부를 포함하는 수준의 BDA 전략도 수립되고 고위 임원으로부터 지원과 지시도 받게 된다. 전사적 성과측정 도구와 방법론이 제정되어서 투자의사결정에 영향을 주게 된다.

Stage 5— Optimized: 조직에서는 지속적이면서도 사전 조율된 전사적 BDA process 개선작업과 가치실현이 진행된다. 전사적으로 절차를 정의하는 전략지침이 만들어지고 강력한 경영진의 지원을 받는다..

## (2) Sprenger의 데이터 성숙모델

한편 미국의 Markus Sprenger는 BI 관점 (즉, 기업에서의 데이터를 활용한 의사결정이라는 관점)에서 데이터의 성숙모델을 다음과 같이 정의하였다.[3]

1단계: 적절히 이용할만한 데이터가 없다

데이터가 축적되지 않거나 아예 데이터에 대한 관리기준조차 정립되지 않은 상태. 그러나 오늘날 이 단계에 머무르고 있는 조직은 거의 없다. (존재할 수가 없다)

2단계: 데이터가 폭주한다

데이터의 폭주에 시달리고 있다. 내부는 물론 외부에서도 많은 데이터가 발생하고 있지만 이를 분석해서 유용한 정보로 만들어 낼 줄은 모르는 상태이다.

직원은 특정 사안이 발생했을 때 관련 데이터를 (즉, 정확한 데이터를) 찾는데 오랜 시간을 소모해서 정작 분석할 여유를 가지지 못한다. 따라서 정작 의사결정을 할 때는 직관에 의해서 (감에 의해서) 결정을 하고 객관적 데이터가 뒷받침되지는 못한다. 따라서 전략의 수립 내지 자원의 전략적 배분에서도 합리성이 극대화되지 못한다.

무엇보다 어려운 것은 의사결정의 소재가 될 정확하고 유용한(relevant) 정보의 원천(source)를 확인하는 일이다. 여기에는 내부 데이터뿐만 아니라 외부 데이터도 포함된다. 그리고 일단 데이터 소스를 확보하였다면 이를 한 곳으로 모아서 적절하게 관리할 수 있는 장치(절차와 방법론 그리고 필요한 기기)를 구축해야 한다. 즉, 분석을 위한 기반을 구축하는 일이다.

3단계: 의사결정에 유용한 데이터(Right Data)의 발견

이 단계에서 기업은 고급의 데이터를 이용해서 각자 올바른 데이터 모델에 적용한다. 그리고 이때 객관적 데이터를 이용하면서도 상황에 맞도록 (context and relevance) 이를 해석하고 적용할 수 있다.

전사차원에서는 분야별 (또는 제품별, 직능별, ...) 분류기준 (corporate taxonomies )와 metadata를 이용해서 해당 데이터를 분류(categorize)하고 의미있는 방식으로 해석할 줄 알게 된다.

이 단계에서 중요한 요소는 조직 내에서 기술적용과 함께 데이터를 다루는 전반적인 문화의 확산이다. (cultural shift) 즉, 컨텐츠를 사용하는 사람이 동시에 데이터를 생성하는 역할의 중요성을 인식하고 이들 전체를 포괄하는 시야를 가져야 한다는 점이다.

4단계r: 데이터를 예측에 활용

여기서는 단순히 과거 데이터 또는 지난 실적을 분석하는 데에서 한걸음 나아가 예측분석 (predictive analysis)까지 수행할 수 있게 된다. 이 단계가 되어야 시장수요와 고객의 행동이 어떻게 될지를 데이터에 근거하면서도 독창적으로 예측할 수 있게 된다. 예컨대 제약회사나 자동차 회사에서 예측분석 (predictive analysis)을 통해 대 고객 서비스의 질을 향상시키기 위한 예측모델을 수립하거나, 생산품질 향상을 위한 시뮬레이션을 올바르게 할 수 있게 된다.

5단계: 데이터를 전략에 활용

여기서는 전사적인 사업모델이 분석모델(analytical models)에 기초하여 객관적으로 수립되고 이용될 수 있다. 즉, 과거 데이터를 활용하되 미래의 사업모델은 분석과 예측이 결합된 미래지향적인 것이 될 수 있다.

한가지 중요한 점은 그때 그때의 유행에 휩쓸리지 않기 위해서는 다양한 예측모델이 개발되고 이용되더라도 한쪽 발은 과거 데이터에 대한 분석작업 (예: data mining)이 지속적으로 추진되어야 한다는 점이다.

이처럼 분석모델, 예측모델이 하나로 통합되어 risks와 opportunities를 가급적 빨리 분류하고 대응방안을 수립 집행하도록 해야 한다.

# 3. 프로젝트 방법론

아직 추진 사례가 충분히 축적되지 않아서 실천적 지침이 많이 제시되지는 않고 있다. 여기서는 hadoopshere에서 제시한 방법론을 소개한다. [4]

추가 사항:

1) Agile 기법[5]을 제안한다. 여기에는 Scrum 또는 Kanban 기반의 산출물 전달기법이 포함된다.

2) 위 방법론은 소프트웨어 솔루션 그 자체의 개발프로젝트는 물론 서비스 상품에 대한 프로젝트에도 적용이 가능하다.

3) 한편 위 방법론은 Gartner 보고서의.빅데이터 분석 프로세스 (Big Data Analytics Process)를 참조 및 수정하여 만들어졌다.

[1] Big Data Analytics: Future Architectures, Skills and Roadmaps for the CIO. (2011/9)

[2] IDC Maturity Model: Big Data and Analytics — A Guide to Unlocking Information Assets, Mar 2013 (Doc # 239771 )

[3] http://www.b-eye-network.com/view/15105

[4] http://www.hadoopsphere.com/2012/11/delivery-methodology-for-hadoop-projects.html



[5] 소프트웨어 개발방법론의 하나로서 기본적으로 "S/W를 빨리 개발하여 업무에 적용한다는 것"이다. 기존의 단계별 완성을 중시하던 폭포모델의 문제점에 대한 반성에서 출발하였다.

Agile 방법론에서는 짧은 개발주기를 반복(iterate)하고 의도된 대로 진행되는지를 수시로 확인하도록 하면서 동시에 실시간 커뮤니케이션을 강조한다.

세부적으로는 다음의 방법론을 포함한다.

(1) RUP (Rational Unified Process) (2) XP (Extreme Programming) (3) SCRUM

출처: <http://www.openwith.net/?p=565>

# Quiz

2017년 12월 1일 금요일 오전 10:46

Main 구현 시 사용했던 Job 클래스를 이용하여 실 운항된 월별 데이터를 출력하시오

힌트
분석표를 보면 운항취소여부가 있다
이 코드를 이용하여 운항 취소된 부분은 연산에서 제외할 것이다.

# AlReportParser

2017년 12월 1일 금요일 오전 11:27

```
package com.care.sort.nonsort;

import org.apache.hadoop.io.Text;

public class AlReportParser {
	private String year;
	private int month;
	private int cancelled;

	public AlReportParser() {}
	public AlReportParser(Text txt) {
		String []airData = txt.toString().split(",");
		year = airData[0];
		month = Integer.parseInt(airData[1]);
		cancelled = Integer.parseInt(airData[21]);
	}
	public String getYear() {
		return year;
	}
	public int getMonth() {
		return month;
	}
	public int getCancelled() {
		return cancelled;
	}
}
```

# MyMapper

2017년 12월 1일 금요일 오전 11:27

```
package com.care.sort.nonsort;

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 MyMapper extends Mapper<LongWritable, Text, Text, IntWritable>{
	@Override
	public void map(LongWritable key, Text value, Context ctx) throws IOException,
	InterruptedException {
		AlReportParser parser = new AlReportParser(value);

		if(parser.getCancelled()==0)
			ctx.write(new Text(parser.getYear()+"."+parser.getMonth()), new IntWritable(1));
	}
}
```

# MyReducer

2017년 12월 1일 금요일 오전 11:27

```
package com.care.sort.nonsort;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable>{
	@Override
	protected void reduce(Text key, Iterable<IntWritable> values, Context ctx) throws
	IOException, InterruptedException {
		int cnt=0;

		for(IntWritable value:values) {
			cnt += value.get();
		}

		ctx.write(key, new IntWritable(cnt));
	}
}
```

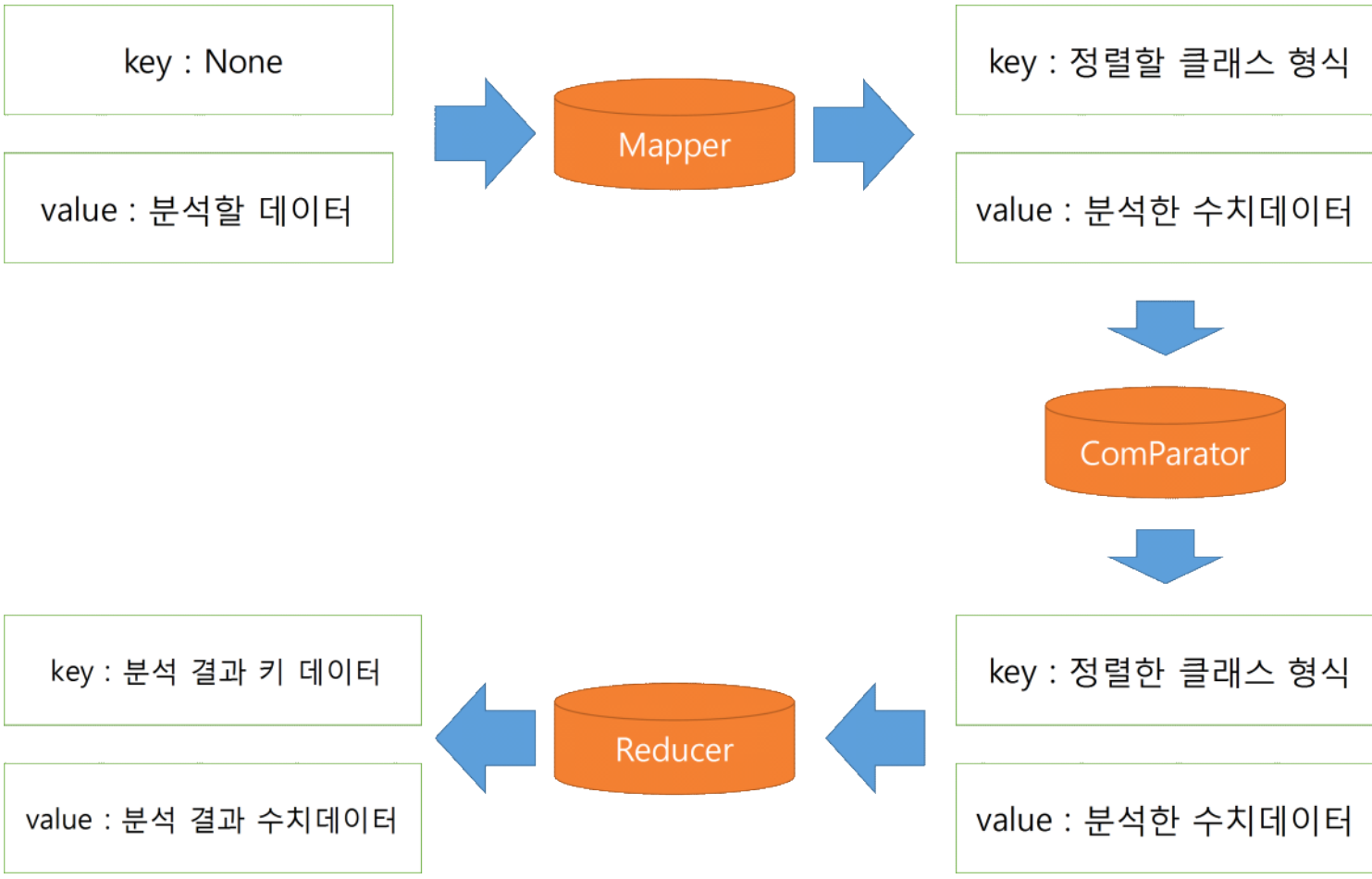
# AirlineMain

2017년 12월 1일 금요일 오전 11:28

```
package com.care.sort.nonsort;

import org.apache.hadoop.conf.Configuration;
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.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class AirlineMain extends Configured implements Tool{
	public static void main(String[] args) throws Exception {
		int exitCode = ToolRunner.run(new Configuration(), new AirlineMain(), args);
		System.exit(exitCode);
	}

	@Override
	public int run(String[] arg0) throws Exception {

		Job job = Job.getInstance(getConf(), "NonSort");

		FileInputFormat.addInputPath(job, new Path(arg0[0]));
		FileOutputFormat.setOutputPath(job, new Path(arg0[1]));

		job.setJarByClass(AirlineMain.class);
		job.setMapperClass(MyMapper.class);
		job.setReducerClass(MyReducer.class);
		job.setInputFormatClass(TextInputFormat.class);
		job.setOutputFormatClass(TextOutputFormat.class);

		job.setOutputKeyClass(Text.class);
		job.setOutputValueClass(IntWritable.class);

		job.waitForCompletion(true);

		return 0;
	}

}
```

# 정렬

2017년 12월 1일 금요일 오전 11:30



# MyMapper 수정

2017년 12월 1일 금요일 오후 2:41

```
package com.care.sort.sort;

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 MyMapper extends Mapper<LongWritable, Text, PartSort, IntWritable>{
	@Override
	public void map(LongWritable key, Text value, Context ctx) throws IOException,
	InterruptedException {
		//AlReportParser parser = new AlReportParser(value);
		PartSort data = new PartSort(value);

		/*data.setYear(parser.getYear());
		data.setMonth(parser.getMonth());*/
		if(data.getCancelled()==0)
			ctx.write(data, new IntWritable(1));
	}
}
```

AlReportParser을 PartSort가 상속받아 AlReportParser 대신 처리 가능

처리 결과를 PartSort를 변경하였음으로 Mapper의 출력 키를 PartSort로 변경

# MyReducer

2017년 12월 1일 금요일 오후 2:44

```
public class MyReducer extends Reducer<PartSort, IntWritable, Text, IntWritable>{
	@Override
	protected void reduce(PartSort key, Iterable<IntWritable> values, Context ctx) throws
	IOException, InterruptedException {
		int cnt=0;

		for(IntWritable value:values) {
			cnt += value.get();
		}

		ctx.write(new Text(key.getYear()+"."+key.getMonth()), new IntWritable(cnt));
	}
}
```

Map에서 전달된 데이터가 PartSort 형식임으로 변경

# PartSort

2017년 12월 1일 금요일 오후 2:45

```
package com.care.sort.sort;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableUtils;

public class PartSort extends AlReportParser implements WritableComparable<PartSort> {
	public PartSort() {	}
	public PartSort(Text txt) {super(txt);}

	@Override
	public void readFields(DataInput in) throws IOException {
		// TODO Auto-generated method stub
		super.setYear( WritableUtils.readString(in) );
		super.setMonth( in.readInt() );
	}

	@Override
	public void write(DataOutput out) throws IOException {
		WritableUtils.writeString(out, super.getYear());
		out.writeInt(super.getMonth());
	}

	@Override
	public int compareTo(PartSort data) {
		int result = super.getYear().compareTo(data.getYear());
		if (0 == result)
			return super.getMonth() - data.getMonth();
		return result;
	}
}
```

# PartSortComparator

2017년 12월 1일 금요일 오후 2:45

```
package com.care.sort.sort;

import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class PartSortComparator extends WritableComparator{
	protected PartSortComparator() {
		super(PartSort.class, true);
	}

	@SuppressWarnings("rawtypes")
	@Override
	public int compare(WritableComparable obj1, WritableComparable obj2) {
		PartSort data1 = (PartSort)obj1;
		PartSort data2 = (PartSort)obj2;

	   return data1.compareTo(data2);
	}
}
```

# AirlineMain

2017년 12월 1일 금요일 오후 2:45

```
package com.care.sort.sort;

import org.apache.hadoop.conf.Configuration;
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.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class AirlineMain extends Configured implements Tool{
	public static void main(String[] args) throws Exception {
		int exitCode = ToolRunner.run(new Configuration(), new AirlineMain(), args);
		System.exit(exitCode);
	}

	@Override
	public int run(String[] arg0) throws Exception {

		Job job = Job.getInstance(getConf(), "Sort");

		FileInputFormat.addInputPath(job, new Path(arg0[0]));
		FileOutputFormat.setOutputPath(job, new Path(arg0[1]));

		job.setJarByClass(AirlineMain.class);
		job.setSortComparatorClass(PartSortComparator.class);

		job.setMapOutputKeyClass(PartSort.class);
		 job.setMapOutputValueClass(IntWritable.class);

		job.setMapperClass(MyMapper.class);
		job.setReducerClass(MyReducer.class);

		job.setInputFormatClass(TextInputFormat.class);
		job.setOutputFormatClass(TextOutputFormat.class);

		job.setOutputKeyClass(Text.class);
		job.setOutputValueClass(IntWritable.class);

		job.waitForCompletion(true);

		return 0;
	}

}
```

# 그룹 정렬

2017년 12월 1일 금요일 오후 2:50

그룹을 이용하여 정렬을 정의할 수 있다.

# GroupKeyComparator

2017년 12월 4일 월요일 오전 10:40

```
package com.care.sort.Group;

import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class GroupKeyComparator extends WritableComparator {

	protected GroupKeyComparator() {
		super(PartSort.class, true);
	}

	@SuppressWarnings("rawtypes")
	@Override
	public int compare(WritableComparable w1, WritableComparable w2) {
		PartSort k1 = (PartSort) w1;
		PartSort k2 = (PartSort) w2;

		// 연도값 비교
		return k1.compareTo(k2);
	}
}
```

# AirlineMain

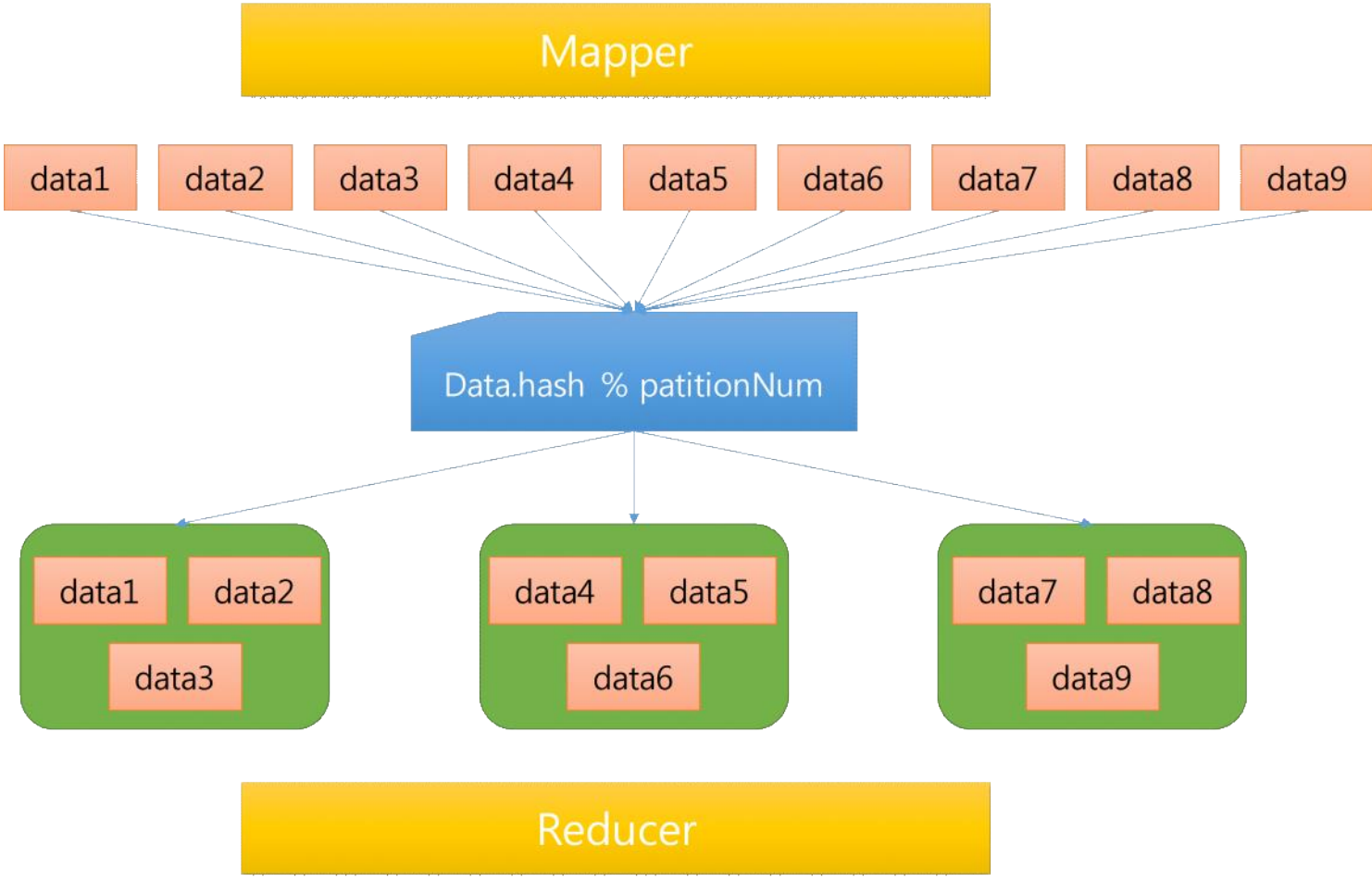
2017년 12월 4일 월요일 오전 10:41

```
package com.care.sort.Group;

import org.apache.hadoop.conf.Configuration;
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.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class AirlineMain extends Configured implements Tool{
	public static void main(String[] args) throws Exception {
		int exitCode = ToolRunner.run(new Configuration(), new AirlineMain(), args);
		System.exit(exitCode);
	}

	@Override
	public int run(String[] arg0) throws Exception {

		Job job = Job.getInstance(getConf(), "ArrDelay");

		FileInputFormat.addInputPath(job, new Path(arg0[0]));
		FileOutputFormat.setOutputPath(job, new Path(arg0[1]));

		job.setJarByClass(AirlineMain.class);
		job.setGroupingComparatorClass(GroupKeyComparator.class);
		//job.setSortComparatorClass(PartSortComparator.class);

		job.setMapOutputKeyClass(PartSort.class);
	  job.setMapOutputValueClass(IntWritable.class);

		job.setMapperClass(MyMapper.class);
		job.setReducerClass(MyReducer.class);

		job.setInputFormatClass(TextInputFormat.class);
		job.setOutputFormatClass(TextOutputFormat.class);

		job.setOutputKeyClass(Text.class);
		job.setOutputValueClass(IntWritable.class);

		job.waitForCompletion(true);

		return 0;
	}

}
```

# 파티셔너

2017년 12월 4일 월요일 오전 10:43

Patitioner는 Reducer가 동작되기 전 해쉬를 이용한 그룹화로 분석 속도를 향상시킬 수 있다.



# GroupKeyPartitioner

2017년 12월 4일 월요일 오전 10:40

```
package com.care.sort.Group;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Partitioner;

public class GroupKeyPartitioner extends Partitioner<PartSort, IntWritable> {

  @Override
  public int getPartition(PartSort key, IntWritable val, int numPartitions) {
    int hash = key.getYear().hashCode();
    int partition = hash % numPartitions;
    return partition;
  }
}
```

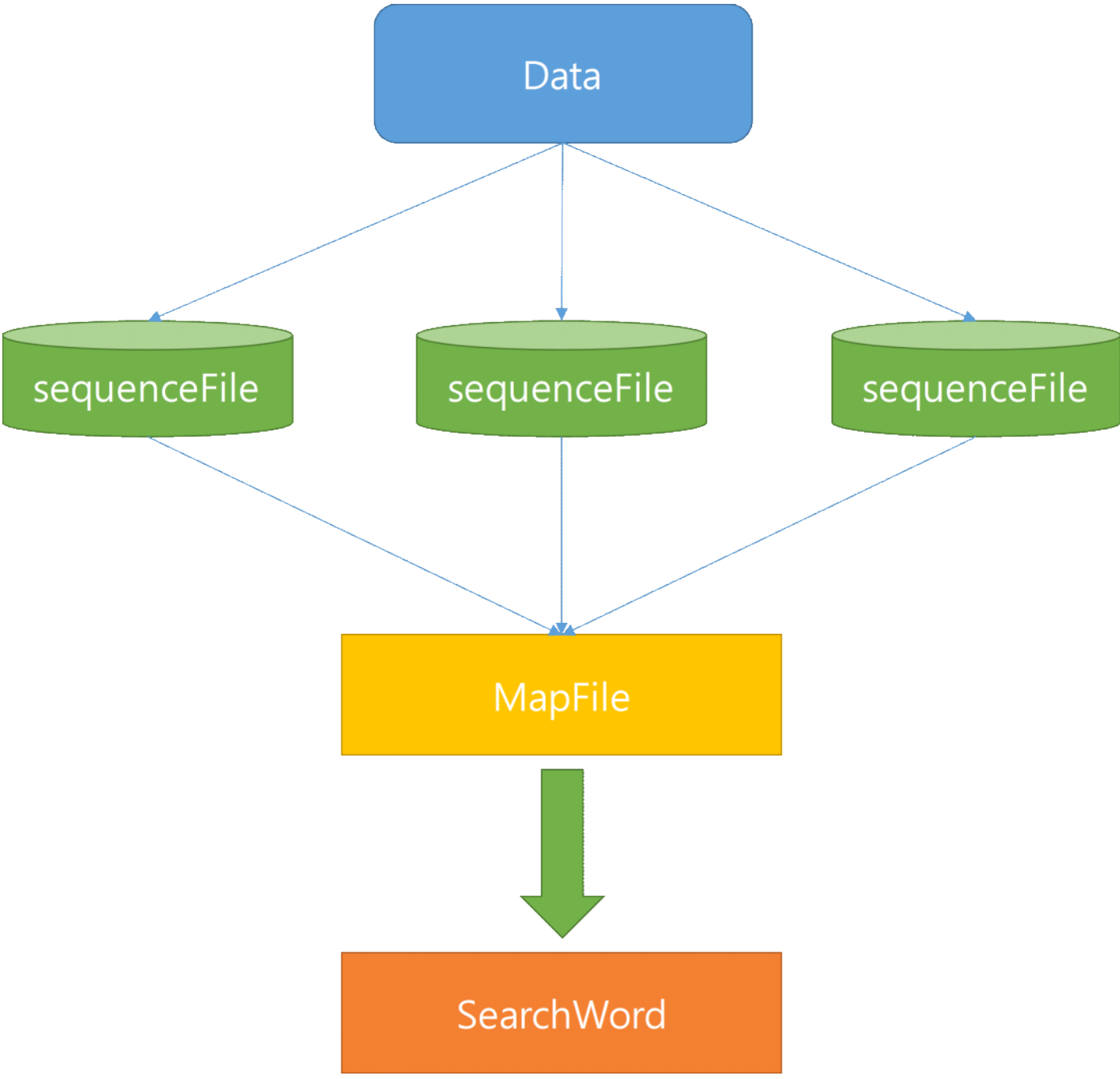
# AirlineMain

2017년 12월 4일 월요일 오전 10:44

```
package com.care.sort.Patitioner;

import org.apache.hadoop.conf.Configuration;
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.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class AirlineMain extends Configured implements Tool{
	public static void main(String[] args) throws Exception {
		int exitCode = ToolRunner.run(new Configuration(), new AirlineMain(), args);
		System.exit(exitCode);
	}

	@Override
	public int run(String[] arg0) throws Exception {

		Job job = Job.getInstance(getConf(), "ArrDelay");

		FileInputFormat.addInputPath(job, new Path(arg0[0]));
		FileOutputFormat.setOutputPath(job, new Path(arg0[1]));

		job.setJarByClass(AirlineMain.class);
		job.setGroupingComparatorClass(GroupKeyComparator.class);
		job.setPartitionerClass(GroupKeyPartitioner.class);
		//job.setSortComparatorClass(PartSortComparator.class);

		job.setMapOutputKeyClass(PartSort.class);
	  job.setMapOutputValueClass(IntWritable.class);

		job.setMapperClass(MyMapper.class);
		job.setReducerClass(MyReducer.class);

		job.setInputFormatClass(TextInputFormat.class);
		job.setOutputFormatClass(TextOutputFormat.class);

		job.setOutputKeyClass(Text.class);
		job.setOutputValueClass(IntWritable.class);

		job.waitForCompletion(true);

		return 0;
	}

}
```

# 부분정렬

2017년 12월 4일 월요일 오후 1:04



# AlReportParser

2017년 12월 4일 월요일 오후 1:04

```
package com.care.sort.partialSort;

import org.apache.hadoop.io.Text;

public class AlReportParser {
	private String year;
	private int month;
	private int cancelled;
	private int distance = 0;
	private boolean distanceAvailable = true;

	public int getDistance() {
		return distance;
	}

	public void setDistance(int distance) {
		this.distance = distance;
	}

	public boolean isDistanceAvailable() {
		return distanceAvailable;
	}

	public void setDistanceAvailable(boolean distanceAvailable) {
		this.distanceAvailable = distanceAvailable;
	}

	public void setYear(String year) {
		this.year = year;
	}

	public void setMonth(int month) {
		this.month = month;
	}

	public void setCancelled(int cancelled) {
		this.cancelled = cancelled;
	}

	public AlReportParser() {
	}

	public AlReportParser(Text txt) {
		String[] airData = txt.toString().split(",");
		year = airData[0];
		month = Integer.parseInt(airData[1]);
		cancelled = Integer.parseInt(airData[21]);
```

```
            // 항공기 출발 지연 시간 설정
            if (!"NA".equals(airData[18])) {
                distance = Integer.parseInt(airData[18]);
            } else {
                distanceAvailable = false;
            }
        }

        public String getYear() {
            return year;
        }

        public int getMonth() {
            return month;
        }

        public int getCancelled() {
            return cancelled;
        }
}
```

# SequenceFileCreator

2017년 12월 4일 월요일 오후 1:06

```
package com.care.sort.partialSort;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.SequenceFile.CompressionType;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import java.io.IOException;

public class SequenceFileCreator extends Configured implements Tool {

	static class DistanceMapper extends MapReduceBase implements Mapper<LongWritable,
	Text, IntWritable, Text> {
		private IntWritable outputKey = new IntWritable();

		public void map(LongWritable key, Text value, OutputCollector<IntWritable, Text>
		output, Reporter reporter)
				throws IOException {

			AlReportParser parser = new AlReportParser(value);
			if (parser.isDistanceAvailable()) {
				outputKey.set(parser.getDistance());
				output.collect(outputKey, value);
			}
		}
	}

	public int run(String[] args) throws Exception {
		JobConf conf = new JobConf(SequenceFileCreator.class);
		conf.setJobName("SequenceFileCreator");

		conf.setMapperClass(DistanceMapper.class);
		conf.setNumReduceTasks(0);

		// 입출력 경로 설정
		FileInputFormat.setInputPaths(conf, new Path(args[0]));
		FileOutputFormat.setOutputPath(conf, new Path(args[1]));

		// 출력 포맷을 SequenceFile로 설정
		conf.setOutputFormat(SequenceFileOutputFormat.class);
		// 출력 키를 항공 운항 거리(IntWritable)로 설정
		conf.setOutputKeyClass(IntWritable.class);
```

```
            // 출력 값을 라인(Text)으로 설정
            conf.setOutputValueClass(Text.class);

            // 시퀀스 파일 압축 포맷 설정
            SequenceFileOutputFormat.setCompressOutput(conf, true);
            SequenceFileOutputFormat.setOutputCompressorClass(conf, GzipCodec.class);
            SequenceFileOutputFormat.setOutputCompressionType(conf,
            CompressionType.BLOCK);

            JobClient.runJob(conf);

            return 0;
        }

        public static void main(String[] args) throws Exception {
            int res = ToolRunner.run(new Configuration(), new SequenceFileCreator(), args);
            System.out.println("MR-Job Result:" + res);
        }
}
```

```
hadoop jar fileOutput.jar com.care.sort.partialSort.SequenceFileCreator /airdata/*.csv /airdata/Ex06
hadoop fs -ls /airdata/Ex06
```

# MapFileCreator

2017년 12월 4일 월요일 오후 1:18

```
package com.care.sort.partialSort;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.SequenceFile.CompressionType;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class MapFileCreator extends Configured implements Tool {

	public static void main(String[] args) throws Exception {
		int res = ToolRunner.run(new Configuration(), new MapFileCreator(), args);
		System.out.println("MR-Job Result:" + res);
	}

	public int run(String[] args) throws Exception {
		JobConf conf = new JobConf(MapFileCreator.class);
		conf.setJobName("MapFileCreator");

		// 입출력 경로 설정
		FileInputFormat.setInputPaths(conf, new Path(args[0]));
		FileOutputFormat.setOutputPath(conf, new Path(args[1]));

		// 입력 데이터를 SequenceFile로 설정
		conf.setInputFormat(SequenceFileInputFormat.class);
		// 출력 데이터를 MapFile로 설정
		conf.setOutputFormat(MapFileOutputFormat.class);
		// 출력 데이터의 키를 항공 운항 거리(IntWrtiable)로 설정
		conf.setOutputKeyClass(IntWritable.class);

		// 시퀀스 파일 압축 포맷 설정
		SequenceFileOutputFormat.setCompressOutput(conf, true);
		SequenceFileOutputFormat.setOutputCompressorClass(conf, GzipCodec.class);
		SequenceFileOutputFormat.setOutputCompressionType(conf,
		CompressionType.BLOCK);

		JobClient.runJob(conf);

		return 0;
	}

}
```

```
hadoop jar fileOutput.jar com.care.sort.partialSort.MapFileCreater /airdata/Ex06 /airdata/Ex07
hadoop fs -ls /airdata/Ex07
hadoop fs -ls /airdata/Ex07/part-00000
```

# SearchValueList

2017년 12월 4일 월요일 오후 1:22

```
package com.care.sort.partialSort;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.MapFile.Reader;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapred.MapFileOutputFormat;
import org.apache.hadoop.mapred.Partitioner;
import org.apache.hadoop.mapred.lib.HashPartitioner;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class SearchValueList extends Configured implements Tool {

	public static void main(String[] args) throws Exception {
		int res = ToolRunner.run(new Configuration(), new SearchValueList(), args);
		System.out.println("MR-Job Result:" + res);
	}

	public int run(String[] args) throws Exception {
		Path path = new Path(args[0]);
		FileSystem fs = path.getFileSystem(getConf());

		// MapFile 조회
		Reader[] readers = MapFileOutputFormat.getReaders(fs, path, getConf());

		// 검색 키를 저장할 객체를 선언
		IntWritable key = new IntWritable();
		key.set(Integer.parseInt(args[1]));

		// 검색 값을 저장할 객체를 선언
		Text value = new Text();

		// 파티셔너를 이용해 검색 키가 저장된 MapFile 조회
		Partitioner<IntWritable, Text> partitioner = new HashPartitioner<IntWritable, Text>();
		Reader reader = readers[partitioner.getPartition(key, value, readers.length)];

		// 검색 결과 확인
		Writable entry = reader.get(key, value);
		if (entry == null) {
			System.out.println("The requested key was not found.");
		}

		// MapFile을 순회하며 키와 값을 출력
```

```
                IntWritable nextKey = new IntWritable();
                do {
                        System.out.println(value.toString());
                } while (reader.next(nextKey, value) && key.equals(nextKey));

                return 0;
        }
}
```

```
hadoop jar fileOutput.jar com.care.sort.partialSort.SearchValueList /airdata/Ex07 100 | head -10
```

# 전체정렬

2017년 12월 4일 월요일 오후 3:03

# SequenceFileTotalSort

2017년 12월 4일 월요일 오후 3:03

```
package com.care.sort.allSort;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.SequenceFile.CompressionType;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.mapred.lib.InputSampler;
import org.apache.hadoop.mapred.lib.TotalOrderPartitioner;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import java.net.URI;

@SuppressWarnings("deprecation")
public class SequenceFileTotalSort extends Configured implements Tool {

	public static void main(String[] args) throws Exception {
		int res = ToolRunner.run(new Configuration(), new SequenceFileTotalSort(), args);
		System.out.println("MR-Job Result:" + res);
	}

	public int run(String[] args) throws Exception {
		JobConf conf = new JobConf(getConf(), SequenceFileTotalSort.class);
		conf.setJobName("SequenceFileTotalSort");

		conf.setInputFormat(SequenceFileInputFormat.class);
		conf.setOutputFormat(SequenceFileOutputFormat.class);
		conf.setOutputKeyClass(IntWritable.class);
		conf.setPartitionerClass(TotalOrderPartitioner.class);

		// SequenceFile 압축 포맷 설정
		SequenceFileOutputFormat.setCompressOutput(conf, true);
		SequenceFileOutputFormat.setOutputCompressorClass(conf, GzipCodec.class);
		SequenceFileOutputFormat.setOutputCompressionType(conf,
		CompressionType.BLOCK);

		// 입출력 경로 설정
		FileInputFormat.setInputPaths(conf, new Path(args[0]));
		FileOutputFormat.setOutputPath(conf, new Path(args[1]));

		Path inputDir = FileInputFormat.getInputPaths(conf)[0];
		inputDir = inputDir.makeQualified(inputDir.getFileSystem(conf));
		Path partitionFile = new Path(inputDir, "_partitions");
		TotalOrderPartitioner.setPartitionFile(conf, partitionFile);
```

```
            // 샘플 데이터 추출
            InputSampler.Sampler<IntWritable, Text> sampler = new
            InputSampler.RandomSampler<IntWritable, Text>(0.1, 1000,
                        10);
            InputSampler.writePartitionFile(conf, sampler);

            URI partitionUri = new URI(partitionFile.toString() + "#_partitions");
            DistributedCache.addCacheFile(partitionUri, conf);
            DistributedCache.createSymlink(conf);

            JobClient.runJob(conf);

            return 0;
      }
}
```

```
hadoop jar fileOutput.jar com.care.sort.allSort.SequenceFileTotalSort /airdata/Ex06 /airdata/Ex08
hadoop fs -text /airdata/Ex08/part-00000 | head -10
```