Ch 8. 스파크 최적화 및 디버깅

SparkConf로 스파크 설정하기

SparkConf 객체는 새로운 SparkContext를 만들기 위해 필요함.

In [21]:

```
#import pyspark
from pyspark import SparkConf
from pyspark import SparkContext

sc.stop()

# conf를 만든다.
#conf= new SparkConf()
#conf.set("spark.app.name", "My Spark App")
#conf.set("spark.master", local[4])
#conf.set("spark.ui.port", "36000")

conf = (SparkConf().setMaster("local").setAppName("My Spark App").set("spark.ui.port","360
00"))

# 이 설정으로 SparkContext를 만듦
sc = SparkContext(conf = conf)
```

In [1]:

```
lines= sc.parallelize(["holden likes coffee", "panda likes long strings and coffee"])
pairs = lines.map(lambda x: (x.split(" ")[0],x))
result = pairs.filter(lambda keyValue: len(keyValue[1])<20)
result.collect()
```

Out[1]:

```
[('holden', 'holden likes coffee')]
```

실행시에도 가능

```
$ spark-submit --class com.example.MyApp --master local[4] --name "My Spark App" --conf spark.ui.port=36000 myApp.jar
```

혹은 기본값 파일을 만들어 실행도 가능

```
$ spark-submit --class com.example.MyApp --properties-file my-config.conf myApp.jar
```

실행하는 애플리케이션의 SparkConf는 한 번 SparkContext의 생성자에 넘겨지고 나면 수정이 불가능함. 즉 SparkContext가 초기화 되기 전에 결정되어야 함.

동일한 속성 값이 여러 곳에 지정되면

1. SparkConf 객체에 직접적으로 set() 함수를 호출
2. spark-submit에 전달되는 플래그
3. 설정 파일
4. 기본 값

순으로 지정됨

일반적인 스파크 설정값

옵션	기본값	설명
spark.executor.memory (--executor-memory)	512m	익스큐터 프로세서당 메모리
spark.executor.core (--executor-cores),</br> spark.cores.max (--total-executor-cores)	1,</br> (없음)	코어 개수
spark.speculation	false	느리게 실행되는 작업을 다른 노드에 복사해서 실행
spark.storage.block</br> Manager TimeoutintervalMS	45000	익스큐터의 동작 여부를 추적하는 데 쓰이는 제한시간 값
spark.executor.extraJavaOptions</br> spark.executor.extraClasspath spark.executor.extraLibraryPath	(없음)	익스큐터의 JVM 실행 옵션
spark.serializer	org.apache.spark.serilizer.JavaSerilizer	객체들을 직렬화시킬 때에 쓰임
spark.[X].port	(목적에 따라 다름)	스파크 어플이 사용하는 포트 값 지정
spark.eventLog.enabled	false	완료된 작업을 로깅하게 됨
spark.eventLog.dir	file:/tmp/spark-events	이벤트 로깅 저장에 쓰이는 주소

데이터 셔플에 쓰이는 저장 디렉토리는 conf/spark-env.sh 안에 SPARK_LOCAL_DIRS 환경변수를 쉼표로 구분된 경로들로 익스포트 해야 함.

실행을 구성하는 것: 작업, 태스크, 작업 단계

input.txt

INFO this is a message with content

INFO this is some other content

INFO Here are more messages

WARN This is warning

ERROR Something bad happened

WARN More details on the bad thing

INFO back to normal messages

In [2]:

```
input = sc.textFile("input.txt")
tokenized = input.filter(lambda line: len(line)>0).map(lambda line: (line.split(" ")))
tokenized.collect()
```

Out[2]:

```
[[u'INFO', u'this', u'is', u'a', u'message', u'with', u'content'],
 [u'INFO', u'this', u'is', u'some', u'other', u'content'],
 [u'INFO', u'Here', u'are', u'more', u'messages'],
 [u'WARN', u'This', u'is', u'warning'],
 [u'ERROR', u'Something', u'bad', u'happened'],
 [u'WARN', u'More', u'details', u'on', u'the', u'bad', u'thing'],
 [u'INFO', u'back', u'to', u'normal', u'messages']]
```

In [10]:

```
# 각 라인의 첫번째 단어를 추출하여 센다.
counts= tokenized.map(lambda words:(words[0],1)).reduceByKey(lambda a,b: a+b)
```

In [11]:

```
input.toDebugString()
```

Out[11]:

```
'(2) input.txt MapPartitionsRDD[3] at textFile at NativeMethodAccessorImpl.j
ava:-2 []₩n |  input.txt HadoopRDD[2] at textFile at NativeMethodAccessorImp
l.java:-2 []'
```

(1) input.txt **MapPartitionsRDD[32]** at textFile at NativeMethodAccessorImpl.java:-2 []\n </br>
| input.txt **HadoopRDD[31**] at textFile at NativeMethodAccessorImpl.java:-2 []

HadoopRDD가 만들어지고 MapPartitionsRDD가 만들어짐

In [13]:

```
counts.toDebugString()
```

Out[13]:

```
'(2) PythonRDD[19] at RDD at PythonRDD.scala:43 [Memory Serialized 1x Replic
ated]₩n |       CachedPartitions: 2; MemorySize: 163.0 B; ExternalBlockStore
Size: 0.0 B; DiskSize: 0.0 B₩n |  MapPartitionsRDD[18] at mapPartitions at P
ythonRDD.scala:374 [Memory Serialized 1x Replicated]₩n |  ShuffledRDD[17] at
partitionBy at NativeMethodAccessorImpl.java:-2 [Memory Serialized 1x Replic
ated]₩n +-(2) PairwiseRDD[16] at reduceByKey at <ipython-input-10-48e80af1eb
e4>:2 [Memory Serialized 1x Replicated]₩n    |  PythonRDD[15] at reduceByKey
at <ipython-input-10-48e80af1ebe4>:2 [Memory Serialized 1x Replicated]₩n
|  input.txt MapPartitionsRDD[3] at textFile at NativeMethodAccessorImpl.jav
a:-2 [Memory Serialized 1x Replicated]₩n    |  input.txt HadoopRDD[2] at tex
tFile at NativeMethodAccessorImpl.java:-2 [Memory Serialized 1x Replicated]'
```

(1) PythonRDD[47] at RDD at PythonRDD.scala:43 []\n

| **MapPartitionsRDD[46]** at mapPartitions at PythonRDD.scala:374 []\n

| **ShuffledRDD[45]** at partitionBy at NativeMethodAccessorImpl.java:-2 []\n

+-(1) **PairwiseRDD[44]** at reduceByKey at :2 []\n

| **PythonRDD[43]** at reduceByKey at :2 []\n

| input.txt **MapPartitionsRDD[32]** at textFile at NativeMethodAccessorImpl.java:-2 []\n

| input.txt **HadoopRDD[31]** at textFile at NativeMethodAccessorImpl.java:-2 []

이 RDD들은 메타데이터만 저장.

In [15]:

```
counts.collect()
```

Out[15]:

```
[(u'INFO', 4), (u'WARN', 2), (u'ERROR', 1)]
```

스케줄러는 액션을 수행할 때 RDD 연산 실행계획을 만든다.</br>
가장 처음 단계부터 부모를 추적해서 물리적 실행계획을 세움.

복잡한 경우 작업단계와 RDD가 1:1로 매칭되지 않음(파이프라이닝, 여러개의 RDD를 합침)

내부 스케줄러는 이미 캐싱된 RDD에 가계도를 제거할 수도 있다. 즉 계산된 결과를 이용하여 앞 부분을 건너뛰기 함. 여러번 재연산되는 것을 방지하는 효과

In [17]:

```
counts.cache()
counts.collect()
# 한단계만 실행
counts.collect()
```

Out[17]:

```
[(u'INFO', 4), (u'WARN', 2), (u'ERROR', 1)]
```

작업단계 그래프가 정의되면 테스크가 만들어지고 내부 스케줄러로 전송됨. 그 후 순차적으로 실행. 데이터 파티션이 테스크 실행, 각 테스크는

1. 데이터 로드(저장장치, RDD, 셔플 결과물)
2. 연산 수행
3. 결과 반환(셔플, 외부 저장장치, 드라이버)

스파크의 실행 단계

1. 사용자 코드가 RDD의 DAG 정의
2. DAG가 액션의 실행계획으로 변환
3. 테스크들이 스케줄링, 클러스터에서 실행

정보 찾기

스파크 웹 UI

기본적으로 4040포트를 통해 가능

Jobs: 진행 상황과 작업 단계, 태스크 등에 대한 수치들

Storage: 영속화된 RDD 정보

excutors: 애플리케이션에 존재하는 익스큐터 목록(Thread Dump 버튼으로 stack trace를 가져올 수 있다.)

Environment: 스파크 설정 디버깅

드라이버와 익스큐터 로그

로그 위치

1. 단독모드에서는 마스터 웹 UI에 직접 표시 (각 작업 노드의 spark_home의 work/ 아래에 저장)
2. 메소스에서 로그는 메소스 슬레이브 노드의 work/ 밑에 저장
3. 얀 모드는 얀의 로그 수집 도구를 사용(yarn logs -applicationid

log4j 기반이므로 conf/log4j.properties.template를 바꿔 log4j.properties 파일을 만든다.(INFO-> WARN or ERROR)

spark-submit --files 플래그로 파일 추가 가능

성능에 관한 핵심 고려사항

병렬화 수준

1. 병렬화 개수가 너무 적으면 스파크가 리소스를 놀림.
2. 너무 많으면 각 파티션에서의 작은 오버헤드라도 누적되면서 성능 문제가 심각해짐.

병렬화 수준 조정

1. 데이터 셔플이 필요한 연산 중에 생성되는 RDD의 병렬화 정도
2. 이미 존재하는 RDD의 재배치(repartition(), coalesce())

In [ ]:

```
#와일드 카드 입력은 수천개 파일이 될 수도 있다.
input= sc.textFile("s3n://log-files/2014/*.log")
input.getNumPartitions()

#필터링
lines=input.filter(lambda line: line.startswich("2014-10-17"))
lines.getNumPartitions()

# 캐싱 전에 RDD 합침
lines=lines.coalesce(5).cache()
lines.getNumPartitions()

lines.count()
```

직렬화 포맷

- 네트워크로 데이터 전송, 디스크에 쓸 때 객체를 직렬화하여 바이너리 포맷으로 변환
- 자바 내장된 직렬화 보다 카이로(Kyro)가 더 빠르고 간편
- spark.serializer를 org.apache.spark.serializer.KyroSerializer로 지정
- spark.kyro.registrationRequred를 true: 직렬화할 클래스를 등록
- 자바의 직렬화 인터페이스를 구현하지 않은 클래스를 참조할 경우 NotSerializableException: spark-submit의 -driver-java-options, --executor-java-options 플래그에 "-Dsun.io.serialization.extendedDebugInfo=true" 같은 옵션을 적용하여 디버깅
- 클래스가 직렬화 구현하도록 하던지, 안되면 자식 클래스를 만들고 자바의 externalizable 인터페이스를 구현하거나 카이로의 직렬화 동작 수정

메모리 관리

메모리 사용 목적

- RDD 저장용: persist(), cache() 호출 시 메모리에 저장(spark.storage.memoryFraction, 60%)
- 셔플 및 집합 연산 버퍼: 셔플 연산시 출력 데이터 저장(spark.shuffle.memoryFraction, 20%)
- 사용자 코드: JVM 힙에 남은 나머지 메모리 사용(20%)

기본 캐싱 동작 요소 개선

1. persist() 사용시 MEMORY_AND_DISK레벨을 적용하면 새로운 파티션이 생길 때 디스크에 쓰게 되어 다시 읽을 수 있음.
2. 기본 자바 객체로 캐싱하는 대신 MEMORY_ONLY_SER, MEMORY_AND_DISK_SER로 직렬화된 개체를 저장(가비지 컬렉션에 걸리는 시간을 줄임)

하드웨어 프로비저닝

1. 메모리,코어
    - spark.executor.memory, supark-submit의 --executor-memory 플래그
    - 얀 모드: spark.executor.cores 등으로 총 개수 지정
    - 메소소, 단독 모드: 스케줄러가 제공(spark.cores.max)
2. 로컬 디스크 볼륨
    - 얀 모드: 얀이 결정
    - 단독 모드: spark-env.sh의 SPARK_LOCAL_DIRS
    - 메소스, 기타: spark.local.dir
3. 메모리 사이징
    - 가비지 컬렉션 작업을 줄이기 위해 작은 익스큐터 메모리를 지정하는 것이 이득일 때가 있다.

In [23]:

```
conf.set("spark.serializer","org.apache.spark.serializer.KyroSerializer")
conf.set("spark.kyro.registrationRequred","true")
#conf = (SparkConf().setMaster("local").setAppName("My Spark App").set("spark.serialize
r","org.apache.spark.serializer.KyroSerializer"))
```

Out[23]:

```
<pyspark.conf.SparkConf at 0x7f14d632c050>
```

In [ ]:

```

```