Abstract

- RDDs(Resilient Destributed Datasets) : 클러스터 기반의 고장방지 분산처리 메모리 추출(추상화) 기법
- RDDs 는 비효율적으로 처리하는 두가지 방식의 컴퓨팅 프레임워크에 동기부여를 받아 개발되었음
⇒iterative algorithm, interactive data mining tools
- 위의 방식들은 메모리를 이용하여 처리하면 성능을 향상시킬 수 있음(by an order of magnitude?)
- 고장방지를 효율적으로 사용하기 위해서는 tranformation 이라는 coarse-grained 방식을 이용하여 메모리를 이용하는 restricted form을 사용
- RDD 방식을 Spark에 적용

1 Introduction

- 기존 방식의 분산처리
  - 대용량 데이터 분석에서는 MapReduce, Dyrad 와 같은 클러스터 컴퓨팅 프레임워크가 사용되어왔으나 병렬처리 기반 연산자를 이용한 것이므로 작업을 분산시키는 부분이나 고장방지에 대한 고려는 없었음
  - 최근의 프레임워크들은 클러스터를 사용하지만 메모리를 분산관리하는 부분이 부족해 비효율적임
  - 중간 결과를 reuse함으로서 외부스토리지를 이용함에 따라 overheads의 문제 발생
  - 기존 application들은 특정 패턴을 따라야 했고 이는 더 일반적인 reuse에 대해서는 부족함.
- RDDs
  - RDDs 는 이러한 부분을 개선하여 더 효율적인 reuse를 가능하게한 새로운 추상화기법임. 고장방지, 여러 중간 결과를 유지할 수 있도록하는 병렬 자료 구조 지원, 데이터 배분을 최적화하기 위한 파티셔닝, 다양한 연산자를 통해 다룰 수 있는 기법을 제공함.
  - RDDs에서효율적인 고장방지를 위해 새로운 인터페이스가 필요함. 클러스터의 소모값을 크게하는 기존의 데이터 복제와 모든 머신을 업데이트 방식에는 한계가 있음.
  - RDDs 는 coarse-grained transformation으로 data lineage를 생성해 효율적인 고장방지를 달성함. transformation 연산을 기억해 그 연산으로 다시 생성하면 문제를 해결할 수 있음. coarse-grained transformation은 실제로 매우 효율적인 연산이 가능하게하는 장점이 있음.
  - Spark 환경에서 스칼라 언어 기반으로 사용하며 빠른 속도를 보장함.

2 Resilient Distrtibuted Datasets(RDDs)

2.1 RDD Abstraction

- RDDs = read-only, partitioned collection of records.
  - RDDs 는 derministic operations(transformation e.g., map, filter)을 이용하여 생성할 수 있는데 안정적인 스토리지에 있는 데이터나 다른 RDDs 에서만 생성 가능하다.
  - Transformation은 변형불가능한 RDD를 이용하여 새로운 RDD를 생성하는 것으로 lineage를 만들어 주는 역할.
  - RDDs 는 matrialized될 필요는 없고 생성정보(lineage)만 있으면 된다. matrialized 상태가 아니므로 참조할 수도 없다.
  - 사용자는 persistence 와 partitioning을 통제할 수 있음. 전자는 재사용할 RDD를 정해 메모리에 보관하며, key를 기준으로 partitioning 작업을 요청할 수 있다. 이는 데이터 배치의 최적화에 도움이 된다.

2.2 Spark Programming Interface

- Spark : 언어 통합 API로 RDDs를 구현
  - RDD는 하나의 object로 표현되고, transformation은 method 로 작동함.
  - 안정적인 스토리에서 transformation 으로 RDD(s)를 만들고 action 연산에서 사용할 수 있게됨. action 을 거쳐야 비로소 데이터로 사용하는 것임.
  - 이를 Lazy execution 과정이라고 하는데 transformation으로 쌓아올려진 RDDs들의 pipeline을 최종적으로 action을 통해 실행하고 사용할 수 있게 하는 것.
  - RDDs에 persist 적용 시 우선적으로 재사용할 RDDs를 지정. RDD는 RAM에서 기본 연산을 하지만 용량이 부족하면 디스크로 내리기도 함.

2.2.1 Example: Consol Log Mining

- HDFS로 부터 로그데이터를 받아 분석하는 것인데, line 1 ~3 분석의 object는 모두 RDDs 로 실제로 실행하지 않음. count action을 사용함으로 실제로 실행됨.
- errors = in-memory(RAM), lines = not in-memory. 여기서 추가로 error에 transformation 해서 action 한 명령은 error를 처리하는 파티션에 보내 추가로 실행하도록 함.

2.3 Advantages of the RDDs

- DSM과 비교
  - DSM는 전체 범위에서 임의의 위치를 지정하는 방법을 사용하는 일반적인 분산처리기술이나, 클러스터에 대해서 효율과 고장방지의 관점에서는 좋은 방법이 아님.
  - RDDs는 coarse-grained transformation에 의해서만 생성될 수 있음. 이는 대용량 write 에 있어서는 제한을 걸지만, 효율적인 고장방지를 가능하게 한다. 또한 데이터가 생성되는게 아니므로 overhead 를 활용하지 않고 데이터를 복구할 수 있으며, 문제 발생 시 전체를 보지 않고 RDDs에서 문제가 생긴 부분만 data lineage에서 찾아 복구할 수 있는 장점이 있다. RDDs는 변형할 수 없기 때문에 느리게 돌아가는 노드(straggler)에 대해서 여유를 주고 백업할 수 있게 하는 장점이 있다. 대용량 처리에서는 성능향상을 위해 데이터의 위치에 따라 작업을 배분할 수 있다. 또한 RAM의 용량이 부족하거나 맞지 않는 데이터는 디스크로 내려보내므로 다른 병렬처리 시스템과 비교 시 적어도 비슷한 성능은 보여준다.

2.4 Applications Not Suitable for RDDs

- RDDs는 동일한 연산을 전체에 실행하는 경우에 효율적이고, 고장방지에 적합한 방법이다. 하지만 동시에 일어나는 업데이트가 아닌 웹 어플리케이션이나 크롤러에는 적합하지 않다. 이런 경우는 기존의 방식이 더 적합하다. RDDs는 배치 분석과 동시에 일어나는 분석에 집중하기 때문에 이부분은 고려하지 않는다.

3 Spark Programming Interface

- Spark는 언어가 통합된 API를 기반으로 RDD 추상화를 제공함. 특히 Scala를 사용하는데 간결하고 효율적인 언어이기 때문이다.
- Spark를 사용하기 위해선 Driver Program을 통해 클러스터의 worker들에 연결해야한다. Driver는 여러 RDDs를 정의하고 action을 지정해준다. Spark의 명령은 RDDs 의 lineage를 따른다. worker들은 연산자가 진행되는 동안 RDDs의 파티션들을 RAM에 계속 저장해둔다.

- RDDs는 그자체로 element type으로 모수화 된 정적인 코드 object인데, Scala 언어는 자료형 참조를 지원하기 때문에 자료형을 표시하지 않아도 된다.
- Scala에 RDDs 를 적용하는 것이 간단해보이지만 실제로는 Scala의 closure object가 reflection을 지원해준다는 문제가 있어 추가적인 해결이 필요하다.

3.1 RDD Operations in Spark

- Spark에서는 RDDs의 연산자로 Transformation과 Action을 지원함. Transforamtion은 Lazy operation으로 나중에 처리하게 되며, Action을 통해 실질적으로 계산, 값 도출, 외부 저장소에 데이터 생성을 하게 된다. 추가로 persist 연산을 통해 RAM에 보관할 RDDs를 지정할 수 있다.

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V,V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T],RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)],RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)],RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T],RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T,T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.,* HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

그림 3.1: Spark의 Transformations , Actions

3.2 Example application

3.2.1 Logistic Regression

– 머신러닝의 대부분의 알고리즘이 최적화에 있어 iterative 한 특성을 가지고 있는데 이는 메모리에 저장하면 더 빠른 속도를 보장할 수 있다.

– Logit Reg 에서 최적 가중치 w를 구하기 위해 GD 방식을 사용할때, points RDD에 persist를 지정하고, map&reduce로 gradient를 계산하여 지속적으로 업데이트 해주게 된다. 이런 persist로 메모리에 RDD를 남기면 20배 정도 빠른 결과를 도출할 수있다.

3.2.2 PageRank

– 저 이거 이해가 안가요

4 Representing RDDs

- Representation ⇒ Graph-Based

– RDDs를 추상화 기법으로 사용하기 위한 문제 중하나가 여러 transformation의 흐름을 추적할 수 있는 대표값이 필요하다는것이다.

– 이상적인 RDDs를 실행하는 시스템은 transformation 연산자를 가능한 다양하게 제공하고 사용자가 임의의 방식으로 사용할 수 있도록 하는 것이다.

– 해답으로 간단한 그래프 기반의 대표값을 제시. 이는 Spark에서 최대한 시스템의 구성을 간단하게 하고 스케쥴러에 특별한 로직을 부여하지 않고 transformation을 다양하게 제공하기 위함이다.

– 각각의 RDD가 5가지의 정보를 표시할 수 있는 인터페이스를 고안했다. 데이터셋을 일정한 단위로 쪼갠 파티션의 집합, 상위 RDDs에대한 종속성, 상위 단위를 기반으로 데이터셋을 연산하는 함수, 데이터의 배치와 파티션 계획에 대한 메타정보가 그 내용이다.

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

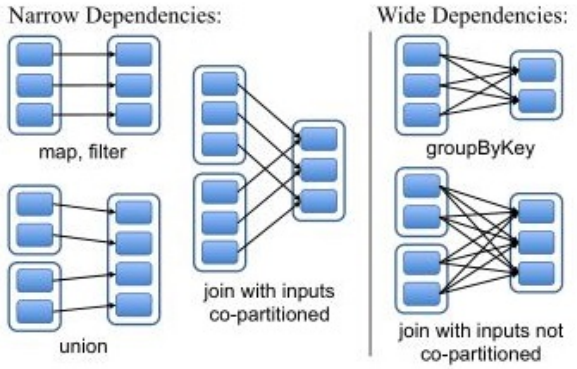

Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

- Dependency ⇒narrow vs. wide
  - 인터페이스 구성에서 흥미로운 문제는 종속성을 표시하는 법이다.
  - narrow dependency : 상위노드 아래 하위노드가 하나일 때(1대 1 대응).상하위 파티션을 연결하여 파이프라인화 된 실행을 가능하게 함. transformation 연산을 연속적으로 수행 가능하게 함.
  - wide dependency : 상위 노드 아래 하위노드가 여러개 일 때(1대 다 대응) 상위 파티션에 대해서 shuffle을 활용한 MapReduce 연산이 필요함. 문제가 생기면 하위노드 여러개에 문제가 생기므로 전체를 다시 실행시켜야하는 번거로움이 생김.
  - 이 종속성을 이용하면 코드를 단순화 하고 스케쥴러의 배분에 대해서도 크게 고민하지 않아도 된다는 장점이 있음.

5 Implementation

- 이 논문에서 스칼라 언어로 스파크를 실행하였으며, Mesos 클러스터 매니저를 이용해 Hadoop, MPI 등과 분산하도록 실행하였음. 각각의 스파크 프로그램은 개별 Mesos 어플리케이션을 실행하였고 그 아래서 driver와 worker, 자원분산들을 Mesos에 의해 실행하였다. 스파크는 기존 Hadoop 시스템의 API를 가지고 데이터를 읽을 수 있으며 스칼라를 수정한 버전을 이용한다.

5.1 Job Scheduling

– 스파크의 스케쥴러는 앞서 말한 RDDs의 graph-based representation을 이용한다. 스케쥴러는 Dryad와 비슷하지만, 퍼시스트가 지정된 RDDs의 파티션을 메모리에 저장시켜놓는다는 점을 추가로 고려하였다.

- 사용자가 action을 실행하면 > 스케쥴러는 실행해야할 stage들의 DAG를 만들기 위해 RDD's의 lineage를 탐색한다. > 각각의 stage들은 좁은 종속성을 가진 transformation들을 가능한 많이 만든다. stage를 나누는 기준은 wide 종속성에서 필요한 셔플 연산자를 이용하거나, 상위 RDD의 연산을 최대한 단축화시킬 수 잇는 미리 계산된 파티션을 이용하기도 한다. 스케쥴러는 원하는 RDD를 만들때 까지 결측된 파티션들을 확인하는 연산도 수행할 수 있다.
- 스케쥴러는 지연 스케쥴링을 이용하여 데이터의 위치에 기반하여 작업을 배정한다. 만약 작업이 한 노드의 메모리에서 연산 가능한 파티션 작업이 필요하면, 그것을 그 노드에 보내준다. 반면RDD가 preffered locations(e.g. HDFS)을 제공한다면 그것을 HDFS에 보내준다.
- wide dependency에서는 고장복구를 단순화하기 위해 상위파티션을 가지고 있는 노드데 대한 중간기록을 남겨놓는다. 만약 작업이 실패하였을 경우에는 해당 스테이지의 상위 부분이 사용가능한 상태인 다른 노드에서 그 작업을 다시 시행한다. 만약앞서 남겨놓은 중간기록이 사라져 몇 스테이지가 사용 불가능하다면 결측된 파티션을 다시 병렬처리하여 생성하도록 보낸다.
- 스파크의 모든 연산이 드라이버 프로그램이 요청한 action에 따라 실행되긴 하지만, lookup 연산을 이용해 클러스터에서 작업이 실행되도록 실험중이라고 함.

5.2 Interpreter Integration

- 스칼라는 Ruby 나 Python과 같이 interactive shell(?)을 제공함. 메모리의 낮은 latency 속에서 사용자가 큰 데이터셋의 쿼리를 불러올 때 interpreter와 상호작용하며 돌릴 수 있게 하기 위함.
- 스칼라 interpreter는 JVM을 이용하여 각각의 라인의 클래스를 컴파일하는 형식을 사용한다.
- 스파크에서 인터프리터를 두가지 방향으로 바꿨다. 먼저 Class shiping은, interpreter는 worker노드가 HTTP에서 bytecode 정보를 가지고 넘어오도록 함. Modified code generation은 코드 생성 로직을 각각의 라인의 인스턴스를 직접 참조하도록 한것을 말한다. (뭔말인지 잘 모르겠음;;)

5.3 Memory Management

- persistent RDDs
  - Persist가 지정된 RDDs의 저장에 3가지 옵션을 사용함. 1) 자바 객체로 메모리에 저장, 2) 시리얼화된 데이터로 메모리에 저장, 3) 디스크에 저장. 각각 빠른 성능, 효율적인 메모리 표시, RDD가 너무 큰 경우 적합하다는 장점이 있다.
- LRU eviction policy
  - 메모리 공간이 제한적일 경우 사용. RDD 생성 시 저장공간이 부족할 때 가장 오래된 RDD의 파티션을 뺀다. (LRU 알고리즘을 그대로 쓰는듯
    - ref. https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)
  - 이에 더해 Persistence priority 라는걸 지정해줄 수 있음.

5.4 Support for Checkpointing

- Data lineage 속에서 복구하기 쉬워지는 건 맞지만, 그 lineage가 길어지고 wide dependency가 포함되면 찾아가는데 시간이 많이 걸리므로 몇 RDDs를 안정된 메모리에 Checkpoint로 만들어준다. narrow dependency에서는 그냥 계산해버리면 되므로 굳이 필요하지 않은 방법이기도 함.

- Spark 에서는 checkpointing 에 대해서 API를 제공하고있지만 사용자가 결정해야한다는 점이 있고, 스케쥴러가 자원과 실행시간을 알고 있어 이를 이용해 자동화하는 문제에대해서 연구하고 있음.
- 다시 말해, read-only RDDs는 checkpoint를 지정을 단순화해준다는 장점이 있음.

6 Evaluation

- 스파크와 RDDs를 적용하여 AWS 인스턴스를 이용하여 성능 평가를 실시. 엄청 빨라짐..
- 속도가 Hadoop이 상대적으로 느려진 이유에 대해 생각하면, 1) 하둡이 쌓을 수 있는 최소 overhead의 수, 2) 데이터를 전송할때 발생하는 Hadoop의 overhead, 3) 자바 객체로 바ㅜ꺼주는데 필요한 Deserialization 소모값 때문이라고 할 수 있음.
- 대부분 빠르다는 설명..

7 Discussion

- RDDs 는 read-only 타입과 coarse-grained 방식이기 때문에 프로그래밍에 적합하지 않아보일 수 있지만 실제로는 매우 적합하며 클러스터 어플리케이션들을 매우 잘 표현함.

7.1 Expressing Existing Programming Models

- RDDs는 지금까지 소개된 클러스터 프로그래밍 모델들을 잘 표현할 수 있음. 결과도 똑같이 나오지만 최적화를 하면서 실행한다는 큰 장점이 있음.
- RDDs의 표현성
  - RDDs의 제한(read-only)이 다른 프로그래밍에 영향을 주는게 없기 때문이다. 많은 프로그램들이 같은 연산을 여러 레코드에 적용하듯이 RDDs도 transformation을 통해 생성되므로 크게 차이가 없음.
  - 여러 RDDs 만들어서 버젼을 표시할 수도 있기 때문에 제한되는 것이 문제가 별로 되지 않고, HDFS와 같은 파일 시스템은 변경을 할수도 없기 때문에 별반 차이가 없음.

7.2 Leveraging RDDs for Debugging

- RDDs의 고장방지는 결과적으로 Debugging 에도 도움이 되었음. data lineage 속에서 RDDs를 다시 생성하고 일정 지점부터 디버깅하며 실행할 수 있음. 이는 overhead를 하나도 기록하지 않으며, RDD의 그래프만 사용하면 된다.

8 Related Work

- Cluster Programming Models
  - 여기에 적용하면 더 효율적이고 확장성이 커지며, 일반화된 실행을 가능하게함.
- Caching Systems
  - In-memory caching을 사용하는 RDDs가 더 빠름..
- Lineage

  - lineage나 provenance information을 찾는 것은 오랫동안 연구 대상이었음. RDDs는 다른 lineage system 과 다르게 복제가 필요없다는 장점이 있어 매우 효율적임

- Relational Databases
  - 개념적으로는 RDBS와 비슷하지만 이들은 fine-grained, read-wrtie access 를 지원함. 이는 overhead를 야기하고 RDDs는 이를 안하도록 하는 획기적인 시스템임.

9 Conclusion

- RDDs는 더 효율적이고 일반적인, 고장방지를 이용한 클러스터 기반 분산처리 기술임.
- RDDs는 기존 분산 처리 프로그램에도 표현가능한 확장성도 가지고 있음.
- coarse-grained transformation 기반으로 lineage를 생성하는 효율적인 고장방지 달성