Neutrino: Revisiting Memory Caching for Iterative Data Analytics

중성미자?: iterative data를 위해 revisiting memory를 cache하자는 느낌

**Abstract**

스파크 - 다른 프레임워크에 비해 반복적인 workload에 엄청난 성능 향상을 제공. RDD를 사용하여 메모리에 데이터를 캐싱하고 분산 클러스터에서 데이터를 계산하는 방식.

이 논문에서는 coarse-grained RDD를 사용하고 데이터 캐싱에서는 discrete 캐시 level만 사용하도록 제한. -> 비효율적인 메모리 사용량과 최적 성능보다 낮아지도록 함. 또한 캐싱 결정을 프로그래머가 해야되서 시스템이 런타임 변경에 적응할 수 없도록 .

이런 문제를 해결하기 위해 RDD partition의 fine-grained 메모리 캐시를 사용하고, 클러스터의 런타임 특성을 기반으로 다양한 in-memory 캐시 레벨을 적응(사용?)하도록 해주는 Neutrino를 사용하자.

1. 프로그래머의 캐싱 결정을 시키지 않고 spark 애플리케이션의의 여러 스테이지의 RDD에 걸친 Data flow graph를 추출해서 data access dependency를 찾아내도록함.
2. dynamic-programming기반 알고리즘을 사용해 클러스터 전반의 캐싱 결정을 유도하고, 다양한 캐싱 수준에서 RDD파티션을 변환하거나 삭제한다…

spark를 확장시켜 Neutrino 프로토타입을 구현하고, 성능 평가를 위해 4가지 machine-learning workload를 사용.

Neutrin는 spark를 기본형으로 사용했을때보다 평균 job execution time을 최대 70%까지 향상시킴!

**I. Introduction**

하둡.. 스파크… RDD… 트랜스포메이션 액션 설명.,.

RDD는 RDD partition하나를 의미하는 각 HDFS block에서의 Action에 따라 load 된 data로 인해 생성된다… 프로그래머는 수동으로 서로다른 여러 action들 사이에서 주어진 cache level에 따라 loaded RDD를 캐싱할 수 있다. 아니면, action이 끝난후 RDD가 메모리에서 삭제된다.

Table1 은 캐시 레벨에 따른 tradeoff(절충점)이다. Table1의 cache level들은 모든 RDD partition을 serialized(compact) 또는 deserialized(fast) 방식으로 로 저장한다. 아니면 메모리 및 디스크의 spark JVM heap의 외부에 저장할 수도 있다.

이 논문에서, coarse-grained cache management(RDD추상화+discrete cache levels)는 비효율적인 메모리 사용량과 낮은 성능을 야기한다는 것을 발견했다.

1. 모든 RDD partition은 런타임 특성(필요한 메모리량, 어떤 worker node에 프리메모리가 얼마나 있는지 등)에 관계없이 모든 worker node에 같은 cache level로 저장된다.
2. 시스템은 RDD(프로그래머에게 노출된 persist & unpersist interface를 사용하는)에 대한 캐싱 결정을 애플리케이션 프로그래머에게 의존한다.

이 때문에 RDD나 partition을 하나의 caching level에서 다른 caching level로 변경하는 것이 불가능해진다.

이런 문제를 spark를 확장한 Neutrino로 해결하자. Neutrino는 serialized or deserialized한 cache level에서 메모리 필요량을 프로그래머가 수동으로 유추할 필요가 없고, 클러스터 configuration을 기반으로 캐싱 결정이 가능하다. 대신 Neutrino가 지원하는 new adaptive cache level을 사용해야 하긴 함. Neutrino는 여러 action/stage에 따라 사용되는 RDD들의 dependency를 파악하는 Data flow graph를 자동으로 추출함.

DFG와 cost model을 이용해서 각 RDD partition의 fine-grained cache level을 계산함.

cost model - dynamic programming algorithm에 따른 것, 총 job 실행 시간을 최적화 하기위한 것. 각각의 worker node에서 클러스터 메모리를 가장 효율적으로 사용하도록 caching decision을 하도록 한다..특정 cache level에서 각 partition들이 필요한 메모리의 양, 각 worker node의 free memory의 양, RDD의 data flow dependency등을 고려함.

이를 통해 Neutrino의 두가지 operation (convert & discard)을 가능하게함

convert - RDD partition의 cache level을 변경

discart - RDD partition을 memory에서 제거

-> 결과적으로 RDD는 cache level에서 partition의 일부분을 차지하면서 HDFS에 캐시되지 않은 상태로 남을 수 있다!

**II. Motivation**

serialization - object structure(string, graph등)을 disk에 기록되도록 byte stream으로 변환

deserialization - 반대

spark에서 HDFS의 데이터는 object를 나타내는 structured RDD로 변환됨.

figure1은 disk의 data set의 크기가 증가함에 따라 serialized & deserialized data 량의 증가를 나타낸 그래프임. deserialized data >> serialized data.(거의 3배)

이는 deserialized data가 byte stream(serialized data)를 제외한 information(그래프의 링크, data type의 length등)를 저장하기 때문.

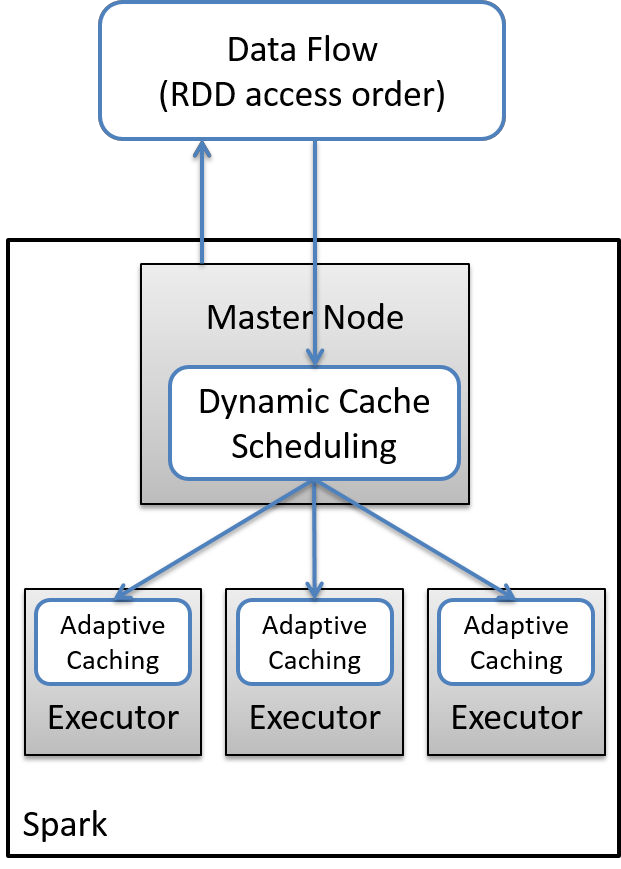
serialized data의 크기는 disk의 data size와 똑같음..

spark의 cache level들은 성능, 메모리 사용량, GC 오버헤드, fault-tolerance 사이에 균형을 이룬다. 젤 많이 사용되는 cache level 3가지 - Mem Deser(spark memory에 deserialized), Mem Ser(spark memory에 serialized), Mem Off(spark 밖의 JVM heap에 serialized)

figure2는 10GB-100GB의 word-count 시킨 결과. 총 캐시 공간은 100GB. Mem Deser는 Mem Ser에 비해 최대 24배 더 성능이 좋음. -> data를 deserialized하는데 overhead가 없으므로.. 그러나 deserialized data의 경우 dataset size의 3배로 증가하기 때문에 40GB 이상의 dataset 부터는 cache 공간을 넘어서서 삭제되기 시작함. cache에서 삭제된 RDD는 HDFS에서 읽어와야 하므로 느려짐. Mem Ser랑 Mem Off는 선형적으로 증가함. Mem Ser가 Mem Off보다 빠른데 이는 Spark외부 heap으로 memory를 복사하는 overhead가 없기때문. 또한 Mem Ser가 50-90GB 사이의 dataset의 경우 cluster에서는 사용하지 않는 메모리를 생성한다는 사실을 관찰

이를 통해 프로그래머는 런타임에 데이터 유형과 cluster의 free memory를 알아내서 가장 적합한 cache level을 결정하기 어려움을 알려줌. 또한 cache level을 coarse-grained 방식으로 결정하면 memory를 가장 효율적으로 사용하지 못함.

**III. System Design**



1. adaptive caching - spark executor에 들어있고, 각 worker node에서 돎.

runtime에 RDD partition의 cache level을 바꾸는 fine-grained cache 관리를 위해 adaptive cache level을 사용.

기존의 spark master의 block manager는 HDFS에서 읽은 모든 partition에 동일한 cache level을 적용시킴(coarse-grained caching).

Neutrino에서는 partition에 cache level을 정의하기 위한 추가적인 structure가 필요함. spark master가 executor에 partition을 할당할때 cache level도 함께 할당.

runtime에 DP scheduling algorithm에 따라 partition의 cache level을 변경하거나 partition을 삭제함. executor는 세가지 operation 사용 - cache, discard, convert.

@cache - partition에 주어진 cache level을 할당

@discard - 바로 삭제되지 않음., asynchronous. 이게 문제가 안되도록 cache나 convert하기 전에 discard된 부분이 ㄹㅇ free memory인지 확인하는 작업 필요. 뭐 암튼 그래서 discard는 overhead가 없다고 함~

@convert - spark와 다르게 cache level을 바꿀 수 있음. Ser <-> Deser 가능. 이를 위해 extra computation이 필요하긴 함.

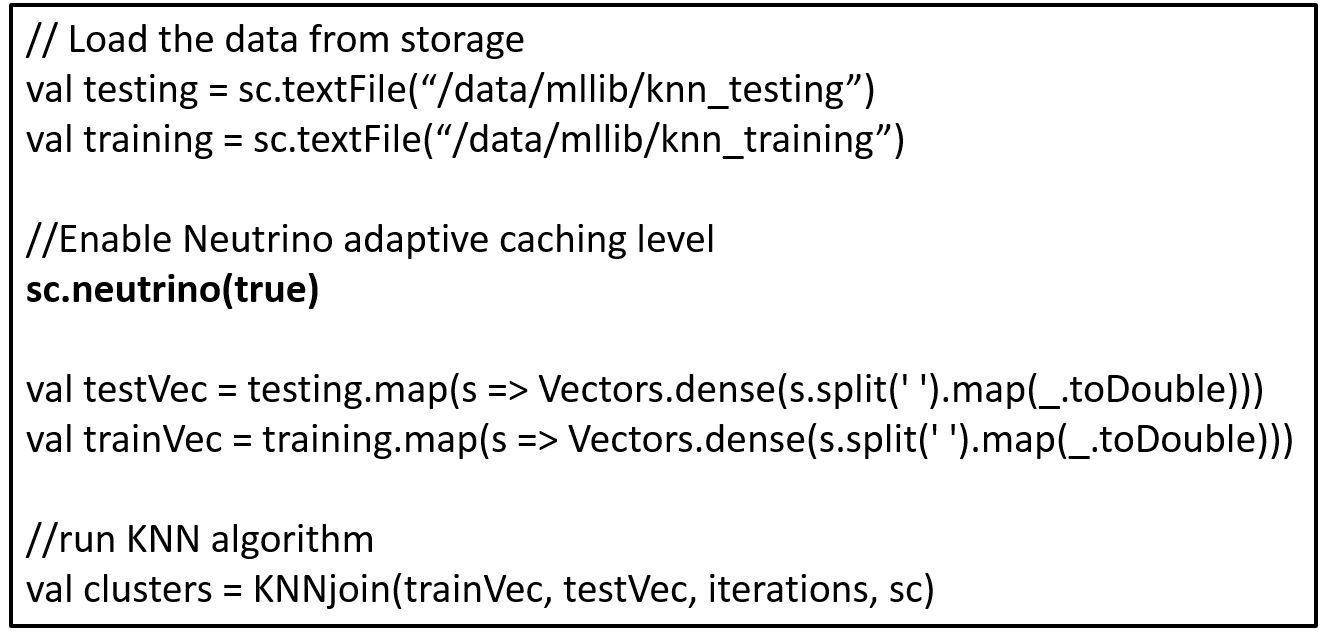
Deser->Ser : 메모리에 더 많은 partition을 load할 수 있음. 다른 partition을 deser로 변환할때 사용할 공간으로 사용가능.

Ser -> Deser : cluster에서 사용되지 않는 메모리를 사용해 성능을 향상시킬 수 있음.

1. data flow graph

RDD의 access order에 따라 DFG를 추출함. 작은 dataset을 실제로 돌려봐서 RDD access order를 얻어낼 수 있다. master node에서 getOrCompute API를 사용하여 얻어냄. access order - job id(key):list of RDD(value)로 이루어짐.

KNN example



testRDD와 trainRDD가 HDFS에서 load되어옴.

map Transformation을 통해 testVec RDD와 trainVec RDD가 생성됨.

KNNjoin은 testVec RDD, trainVec RDD에 Action(or stage?)을 반복적으로 적용하는 함수.

DFG에는 첫번째 stage에는 4개의 RDD가 모두 표현되며 그 이후로 iterative stage에서는 사용되는 2개의 RDD만 포함됨.

1. dynamic cache scheduling - Master node에 있으며 runtime cost model을 사용해서 각 executor의 cache에서 convert & discard를 결정.

Neutrino는 Spark master를 확장시켜 runtime information을 사용해서 cahcing decision을 하도록 함. 각 partition의 상태를 기록하는 partitionStatus Map을 master node에 저장. partition의 block id(key):partition의 ser/deser size, partition의 위치, 현재 cache level, partition의 pending action 등을 저장.

Master는 애플리케이션에서 만들어진 testing, training RDD(dataset)를 사용해 캐싱결정을 함. 두 RDD는 vector로 mapping되고 dataset partition의 위치를 사용해 compute 작업을 함. task들이 서로 다른 worker node들로 스케쥴링 되기 전에 Neutrino의 master는 Dynamic Cache scheduling 알고리즘을 사용해 캐싱결정(convert or discard)을 해 각 RDD partition의 캐싱 레벨을 선택.

Dynamic cache scheduling algorithm

Data flow graph(rdd seq)와 partition status map(pMap)을 사용해 애플리케이션을 실행할 수 있는 최소 시간을 계산한다.

rdd seq[i] = DFG의 i번째 stage/action에서 사용된 RDD list

pMap = 각 partition의 cache level과 위치

dy\_sched 알고리즘 = i단계 이후 다른 partition들에 가능한 연산(convert, discard, cache)의 모든 조합을 탐색하여 i번째 stage의 최소 execution time을 계산함. 불가능한 조합(불충분한 메모리를 캐싱한다거나, 캐시되지 않은 파티션을 변환한다거나 등등)은 빼고 계산함.

Access 함수 = i번째 stage에서 주어진 cache level(pMap)의 경우 data를 읽는데 걸리는 시간을 계산

Executing 함수 = i번째 stage의 맨 끝에 선택된 operation을 적용하는 시간을 계산

**IV. Evaluation**

Mem\_Ser와 Mem\_Deser에서 Neutrino와 Spark를 비교!

4개의 MLlib workload: K-means, KNN, Latent Dirichlet Allocation(LDA), Logistic Regression

-> 3개의 유명한 카테고리(inference 추론, nearest neighbor, regression 회귀)를 포함하도록

-> 또한 한개이상의 RDD를 iteration하도록

Fig7의 세로축은 Neutrino에 비해 Spark의 평균 작업 실행 시간 비율

3가지 시나리오를 사용!

1. Deserialized data size < cluster memory
2. Deserialized data size = cluster memory
3. Serialized data size = cluster memory

대부분의 경우에 Neutrino가 Spark보다 성능좋음.

1. 시나리오 1

Spark Mem-Ser에 비해 45~60% 정도 더 빠름.

-> Neutrino가 모든 partition을 deserialized하고 unused cluster memory를 더 효율적으로 사용하기 때문.

Spark Mem-Deser에 비해 7%정도 느림

-> Dynamic Cache scheduling과 추가적인 cache or convert 연산의 overhead 때문

-> discard 연산은 asynchronous 작업으로 overhead가 없음

1. 시나리오 2

Spark Mem-Ser에 비해 16~35%, Mem-Deser에 비해 5~66% 빠름.

-> Spark Mem-Deser의 memory cache가 누락되기 시작해 HDFS에서 partition을 다시 계산해와야 하기 때문. 그에 비해 Neutrino는 memory가 언제나 cache hit을 하고.

-> Spark Mem-Ser는 job execute중에 data를 deserialized하는 overhead.

1. 시나리오 3

Neutrino는 메모리의 모든 RDD partition을 serialized하므로 Spark Mem-Ser와 거의 동일한 성능을 가짐.

Spark Mem-Deser보다 46~70% 정도 빠름 -> Spark Mem-Deser의 cache miss가 시나리오 2보다 더 자주 생기기 때문에 성능증가가 더 큼.

**V. Conclusion & Future Directions**

Fine-grained caching with a new Dynamic Cache Scheduling Algorithm.

성능은 매우 좋음.

향후에는…

1. non-deterministic execution flow를 초래하는 iterative workload의 dependency도 추적할 수 있는 data flow generation process 만들기
2. pruning technique를 발전시켜 탐사해야될 공간을 줄이고, dynamic cache scheduling algorithm의 fairness를 더 강력하게 보장하도록 하기