HaLoop: Efficient Iterative Data Processing on Large Clusters

Yingyi Bu / Bill Howe / Magdalena Balazinska / Michael D. Ernst
University of Washington

2014104155 채명준

1. motivation

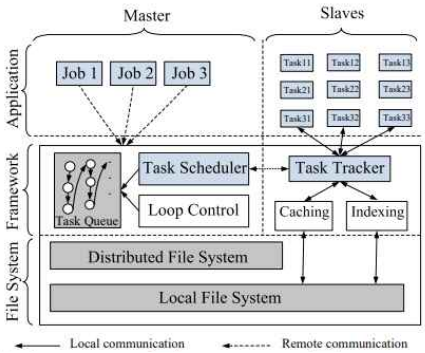
최근 사회적으로 대용량 데이터 처리와 데이터 분석에 대한 수요가 늘어남에 따라 많은 분산처리 프레임워크들이 등장했다. 이러한 분산처리는 Google이 제안한 데이터 처리방식인 Mapreduce를 많이 채택하고 있다. 대표적인 Mapreduce 구현체로는 Hadoop, Dryad, Hive, Pig, HadoopDB등이 있으며, 특히 그 중에서 Hadoop은 사람들에게 가장 많이 알려진 프레임워크이다.

그러나 이렇게 유명한 Hadoop도 데이터를 처리하는 데 있어 단점이 존재했다. Hadoop은 Word count와 같은 단일 반복에 대해서는 효과적인 처리를 제공하지만, 반복적 작업을 지원하지 않기 때문에 반복적인 데이터 처리가 필요한 분야 (ex. Data mining, Web Ranking, Graph Analysis, Model fitting 등)에서는 한계점이 존재하는 것이다.

Hadoop의 이러한 반복적 데이터 처리의 단점을 보안하기 위해 Mahout, Twister, Pregel과 같은 관련 연구가 있었지만, 이들 또한 fundamental한 api를 제공하지 않는 점, 클러스터를 많은 노드로 구성할 수 없는 점 등의 각자의 단점을 가지기 마련이였다.

따라서 우리는 이 논문에서 저수준 API를 제공하며, cache를 초점으로 한 반복적 작업을 효과적으로 처리하는 Hadoop 개량 프레임워크인 Haloop을 소개하고자한다.

2. method or solution



※Haloop Architecture 상속 변경 추가

위에 보는 그림은 Haloop의 Architecture의 구성에 대한 그림이다. Haloop은 기본적으로 분산 컴퓨팅 모델과 Hadoop의 Architecture를 상속받는다. MasterNode와 SlaveNode의 존재, Job 처리 방식 및 데이터 저장소로 HDFS를 사용하는 것 등이 바로 그것이다.

이러한 구조에서 구성을 변경하거나 새로 추가하는 것으로 반복적 프로그래밍을 지원하는데, 큰 분류로 [1)]Loop-aware task scheduling [2)]Cache and Index [3)]API 지원으로 나뉜다.

1) Loop-aware task scheduling

a) Inter-Iteration Locality

이전의 Hadoop에서는 서로 다른 iteration에서 같은 데이터를 처리할 때, 매번 해당 데이터를 HDFS나 Local File System에서 불러왔다. 이는 바로 I/O cost 초래했으며 처리과정에서의 효율성을 저하시켰다. 하지만 Haloop은 처리한 Data를 해당 Node에 Caching해놓고 Task Scheduler로 하여금 해당 데이터를 다시 처리하게 될 경우, Task를 해당 Node에 할당하여 내부에 저장되어있는 데이터를 Load 하게한다.
이것이 바로 Haloop Scheduler의 최종적 목표인 Inter-Iteration Locality로, 다른 Iteration에서 같은 데이터를 사용하는 Mapreduce Task를 같은 Node에 할당하는 것이다.

b) Scheduling Algorithm

```
Input: Node node
// The current iteration's schedule; initially empty
Global variable: Map⟨Node, List⟨Partition⟩⟩ current
// The previous iteration's schedule
Global variable: Map⟨Node, List⟨Partition⟩⟩ previous
 1: if iteration == 0 then
 2:     Partition part = hadoopSchedule(node);
 3:     current.get(node).add(part);
 4: else
 5:     if node.hasFullLoad() then
 6:         Node substitution = findNearestIdleNode(node);
 7:         previous.get(substitution).addAll(previous.remove(node));
 8:         return;
 9:     end if
10:     if previous.get(node).size()>0 then
11:         Partition part = previous.get(node).get(0);
12:         schedule(part, node);
13:         current.get(node).add(part);
14:         previous.remove(part);
15:     end if
16: end if
```

※Scheduling Algorithm Persudo

이러한 Scheduling은 좀 더 효율적으로 동작하기 위한 방식이 정해져 있는데, 그것이 Scheduling Algorithm이다. Slave Node가 Master Node에게 HeartBeat를 보내면, Master Node는 해당 Node가 잘 동작하고 있음을 확인하고, 해당 Node가 이전의 Iteration에서 이미 처리해서 Local에 caching 해놓은 데이터를 사용하는 Task를 할당한다. 만약 해당 노드가 full node로 더 이상 task를 할당받을 수 없으면, 다른 노드에 Task를 할당하여 처리한다. 마약 Cached Data가 사용불가 상태가 되었을 때는, map task의 node 또는 HDFS에서 해당 데이터를 가져와서 처리한다.

2) Cache and Index

: 위에서 언급한 Inter Iteration Locality 덕분에 변하지 않는 데이터의 처리은 하나의 Node의 할당만으로 가능해졌다. 이것이 Cache며, 이 Cached Data에 좀 더 빨리 접근하기 위해서 Indexing 과정을 거친다. 이 때 Caching하는 데이터는 총 세 가지가 있으며, [a)]Reducer Input Cache [b)]Reducer Output Cache [c)]Reducer Output Cache로 나뉜다.

a) Reducer Input Cache

Map Task에서 처리한 Output에 대해서 변하지 않은 불변 데이터가 존재할 경우, Haloop은 모든 Reducer의 Reducer Input을 Caching하고 Index 파일을 생성한다. Cache로 저장할 때, Key와 Value들은 각각 따로 저장되며, 각 Key와 Value는 pointer로 접근한다.

이후 반복에서 해당 Reducer는 불변 데이터가 아닌 데이터들에서만 Mapper Output을 받으며, 해당 Map Key들을 가지는 Value들을 Cached Data에서 찾아서 reduce function에 넣어준다. 이 때 Cached된 데이터는 정렬되어있으며 순차적으로 Disk seek operation에 의해 읽혀지기 때문에 원하는 Cached Data가 맨 마지막에 존재할 경우, 전체 Data를 다 읽어야한다는 단점이 있다.

이러한 Reducer Input Cache를 사용하기 위한 조건이 있는데 바로 매 반복을 거칠 때, Reduce Task들의 개수가 일정하게 유지 돼야한다는 것이다. 만약 RT_0와 RT_i (이 때, RT은 Reducer Task, i는 반복 횟수)의 개수가 다를 경우, Scheduler가 Task를 Node에 할당할 때, 오류가 발생한다. N개의 Cached된 Data를 가진 Node가 있는데, Task가 N-1개면 Task할당이 더 복잡해진다.

b) Reducer Output Cache

Reducer Output Cache는 가장 최근의 결과를 각각의 Reducer node에 Caching하고 Indexing한다. 이는 Fixpoint 평가를 위해서 사용되는데, 여기서 fixpoint란 종료 조건을 말한다. Haloop의 종료 조건은 두개로 나뉘는데 하나는 사용자가 정한 Loop의 횟수를 도달 했을 때 끝나는 것이며, 나머지 하나는 결과의 값이 수렴할 때이다. Fixpoint는 이 때 수렴 종료 조건을 뜻하며 이는 $R_i = R_{i+1}$ (R은 결과 I는 반복 횟수)라는 공식으로 표현된다. Hadoop에서는 이 Fixpoint를 평가하기 위해서 연속된 두 Iteration의 output을 HDFS에 접근해서 MasterNode에서 평가한다. 그러나 Haloop에서는 Node에 저장한 Reducer Output Cache를 이용하여 분산된 환경에서 수렴 조건을 평가하며 각 Reducer의 평가 정보를 Master에서 추합한다.

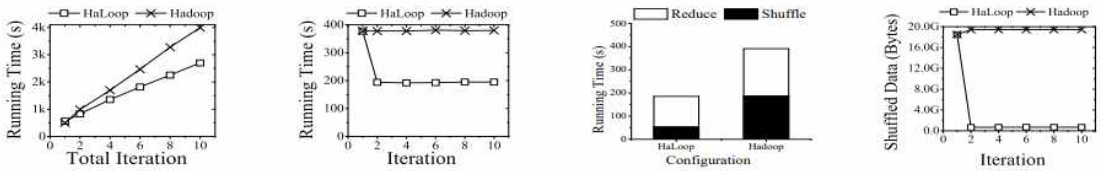
c) Mapper Input Cache

Mapper Input Cache는 초기 반복을 제외한 나머지 반복에서 Data를 local에서 읽고자 하는 목적성을 달성하기 위해 저장하는 Cache이다. 첫 반복에서 Data를 읽은 후, local disk에 저장하게 되면, 이후의 반복에서 HDFS나 local file system에 접근할 필요없이 cached된 데이터를 사용한다.

3. analysis of experimental results (you can attach results)

분석은 Reducer Input Cache, Reducer Ouput Cache, Mapper Input Cache로 각각 나뉘어 실행되었다.

1) Reducer Input Cache

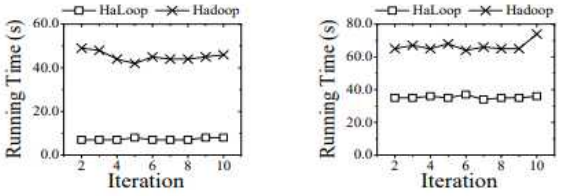


(a) Overall Performance (b) Join Step (c) Cost Distribution (d) Shuffled Bytes

Figure 9: PageRank Performance: HaLoop vs. Hadoop (Livejournal Dataset, 50 nodes)

: Reducer Input Cache 성능 평가는 Decendent Query 및 Page Rank 알고리즘에 대하여 50대의 VM 그리고 90대의 EC2로 구성한 클러스터에서 Live Journal, Free Base, Triple 데이터셋를 처리하는 결과를 보여준다. 모든 경우에서 Haloop의 성능이 Hadoop보다 높게 나왔으며 그 중에서 Decendent Query + Triple이 가장 우수하고, Decendent Query + Live Journal의 성능이 가장 저조하게 나왔다. 이는 Triple이 데이터의 연결도가 가장 적어서, Live Journal가 데이터 복제가 가장 많아서 나온 결과이다.

2) Reducer Ouput Cache

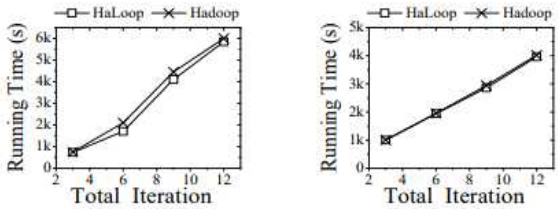


(a) Livejournal, 50 nodes (b) Freebase, 90 nodes

Figure 13: Fixpoint Evaluation Overhead in PageRank: HaLoop vs. Hadoop

: Reducer Ouput Cache 성능 평가는 Page Rank 알고리즘에 대하여 위와 같은 클러스터 환경에서 Live Journal, Free Base 데이터셋으로 진행되었다. 결과적으로 Hadoop보다 Haloop이 40%의 성능 향상을 보였이며, 이는 수렴 종료 조건을 평가하는데 있어서 Fixpoint의 추가적인 Mapreduce의 작업이 없었기에 나타난 결과이다.

3) Mapper Input Cache



(a) Cosmo-dark, 8 nodes (b) Cosmo-gas, 8 nodes

Figure 14: Performance of k-means: HaLoop vs. Hadoop

: Mapper Input Cache 성능 평가는 K-means 알고리즘에 대하여 Commodity 8대의 환경에서 Cosmo-dark, Cosmo-gas를 사용하여 진행되었다. 그래프에서 보다싶이 Hadoop과의 결과적으로 많이 차이 나지는 않지만, 조금 더 성능이 개선된 점을 파악하자면 초기 반복에서만 Local File System, HDFS에서 데이터를 가져왔기 때문이다.

4. opinion or improvement point

Haloop은 Hadoop의 반복적 프로그래밍을 지원하지 않는 단점을 Architecture의 변형 및 추가를 통해서 효과적으로 개선시켰다. 세 개의 Cache Data를 사용하여 I/O COST를 최소화하였으며, 이는 기존 Hadoop보다 1.85배라는 성능의 개선을 가져왔다. 하지만 이러한 Haloop의 Architecture에도 제약조건이 있었으며, 우리는 그것을 Reducer Output Cache에서 볼 수 있었다. Reducer Output Cache를 사용하기 위해서는 항상 동일한 TASK의 개수를 유지해야 하는데, 매 반복을 거칠 수록 사용이 저조한 Task들이 발견될 것이다. 하지만 Cached Data를 가진 노드의 수가 일정하기때문에 해당 Task들을 줄일 수 없으며, 해당 Task들의 지속적인 할당을 받는 Node들은 다른 Node들보다 평균적으로 데이터 처리를 하지 않는 공백이 생길 것이다. Haloop은 I/O COST를 줄여 충분히 훌륭한 성능을 보여주지만, 이러한 Node와 Task의 개수를 줄이게 된다면 좀 더 나은 성능을 기대할 수 있을 것이다. 해당 내용을 개선하는 방법을 강구하기 위해 MasterNode에서 Task 처리 빈도를 Monitoring하여 지속적이 처리의 공백이 있는 Node에 다른 Task의 분할하여 할당하는 방법을 생각했지만, 이는 Reducer 이후에 후속 처리가 한 번 더 필요하다는 점 등의 한계로 완벽하지 않은 방법이라는 판단을 내렸다.