HaLoop:
Efficient Iterative Data Processing on Large Clusters

Yingyi Bu / Bill Howe / Magdalena Balazinska / Michael D.Ernst 2010

2014104124 이규태

○ **motivation**

사회에서 대용량 데이터 처리와 데이터 분석에 대한 수요가 증가함에 따라 MapReduce, Dryad와 같은 데이터를 분산처리 시켜 처리 속도를 향상시킬 수 있는 플랫폼들이 나오게 되었다. 대표적인 예로 Hadoop이 있다. Hadoop은 MapReduce 로서 높은 내고장성과 일반적인 컴퓨터로 클러스터를 구축하여, 저용량으로 대용량 데이터 처리가 가능하다는 장점이 있어 많은 곳에서 사용되고 있다.

그러나 Hadoop에서는 따로 반복 프로그래밍에 대한 지원기능 탑재되어 있지 않았기 때문에, 대용량 데이터를 반복으로 처리해야하는 Page Rank, model fitting, Graph analysis 등의 알고리즘에서는 큰 성능을 낼 수가 없었다.

대용량 데이터의 반복 프로그래밍을 가능하게 하기 위하여 Haloop이 설계되었다. Haloop은 Hadoop을 기반으로 만들어 졌으며, 반복적인 프로그래밍 지원이 가능한 플랫폼으로 Data locality, caching & indexing을 이용하여 반복 프로그래밍에 대해서 Hadoop보다 성능을 1.85배 올렸다.

○ **method or solution**

1) Architecture

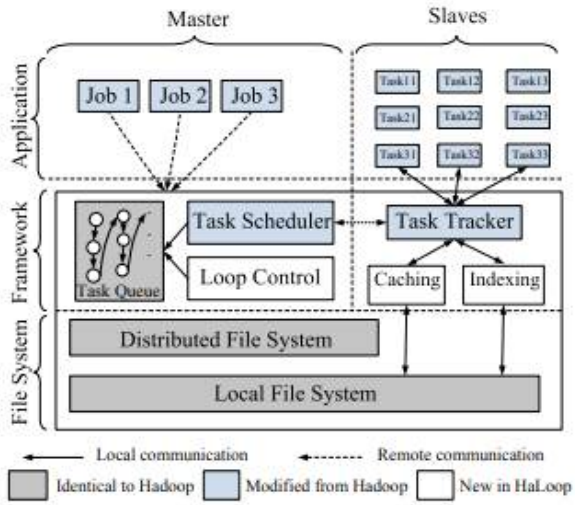

그림 1 Architecture of Haloop

그림 1은 Haloop의 구조를 나타낸다. Haloop은 Hadoop을 기반으로 만들어졌기 때문에 전체적인 구조는 Hadoop과 유사하다. 그러나 반복 프로그래밍을 지원하기 위해서 Loop Control, Caching과 Indexing 기능이 추가 되었으며, 새로운 기능을 제어하기 위해서 Task Scheduler와 Task Tracker를 수정하였다. Haloop은 Loop Control 모듈을 이용하여 유저가 설정한 종료 조건을 충족시킬 때 까지 MapReduce를 반복 수행시킨다. Task Scheduler는 Data Locality를 유지시킬 수 있도록 데이터를 컴퓨터에 할당하는데, 이 작업을 통해 변하지 않는 데이터들을 Cach하고 Indexing 시켜 분석 속도를 높이는 것이 Haloop의 목표이다. 마지막으로 Task Tracker는 slave node를 관리하면서, cach와 index도 관리하는 역할을 한다.

2) Programming Model

$$R_{i+1} = R_0 \cup (R_i \bowtie L)$$

그림 2 Haloop's Programming Model

그림 2는 Haloop에서 효과적으로 동작할 수 있는 프로그램 모델이다. R_0는 초기 결과를 뜻하고, L은 불변 데이터를 뜻한다. Haloop은 불변 데이터들을 캐시데이터로 유지시켜 I/O 비용을 줄인다. 따라서 반복 알고리즘들 중에서도 불변 데이터를 많이 사용하는 프로그램 모델에서 큰 성능을 보인다.

3) Loop-aware task scheduling

```
Task Scheduling
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
```

그림 3 Task scheduling algorithm

Loop-aware task scheduling은 Haloop이 반복적으로 MapReduce 작업을 수행시켜주며, 동일한 머신에서 동일한 데이터를 처리할 수 있도록 데이터를 분배해주는 것을 뜻한다. 동일한 머신에 동일한 데이터를 지속적으로 넣어주는 것을 "Inter-Iteration Locality"를 유지시킨다고 부르는데, Inter-Iteration Locality를 이용해 L을 유지함으로써 Haloop은 I/O 시간과 Data Shuffle 시간을 크게 감소시킬 수 있다.

Inter-Iteration Locality를 유지하기 위해선 Haloop은 매 반복마다 동일한 Reduce Task수를 유지시켜 줘야한다. 그림 3은 Haloop에서 Inter-Iteration Locality를 유지시키기 위해 작성한 스케쥴링 알고리즘의 수도코드다. 데이터를 최대한 이전에 해당 데이터를 처리했던 머신으로 분배시키며, 해당 머신이 일을 받을 수 없을 때, 최대한 근처 노드에 할당함으로서 Locality를 유지시키는 방법을 사용한다.

4) Caching and Indexing

Inter-Iteration Locality 덕분에 Haloop은 L에 접근하는데 하나의 노드만 필요하고, 추후 L을 재사용하여 I/O 비용을 줄이기 위해 해당 데이터들을 local disk에 저장하였으며, 해당 데이터에 빠른 접근을 위해 Caching 과 indexing을 설계하였다. Haloop에서는 총 3개의 Cache를 사용하여 성능을 향상시켰다.

Reducer Input Cache는 L에 대하여 map에서 처리한 결과를 받아야하는 Reduce에 저장해 놓는 것이다. Inter-Iteration Locality로 마스터 노드는 L을 처리한 Map의 위치와 처리 결과를 받을 Reduce를 알고 있기 때문에 해당 데이터는 다시 Map에서 재 업로드 되어 실행될 필요가 없고, Reduce가 결과를 가지고 있어서 shuffle할 필요도 없어서 성능을 향상시킨다. Reducer Input Cache는 L을 많이 사용하는 Page Rank나 Decedent query등에서 많이 사용된다.

Reducer Output Cache는 각각에 Reducer에 이전의 결과 값을 저장해두는 것이다. fixpoint에 도달했을 때 프로그램이 종료되는 Page Rank와 같은 알고리즘에서는 Reducer Output Cache를 이용하여, 각각의 Reducer에서 이전 결과와의 차이를 계산할 수 있다. 마스터 노드는 해당 결과들만 취합하여 반복 작업을 계속 진행할지 결정할 수 있게 되므로 Reducer Output Cache를 이용하지 않았을 때 추가적인 맵리듀스 작업을 해야한다는 걸 고려했을 때 Reducer Output Cache를 이용하는 것이 효과적이다.

Mapper Input Cache는 K-means와 같은 model-fitting알고리즘처럼 mapper에 입력으로 들어가는 데이터가 변화가 없을 때, 해당 데이터를 캐시로 유지함으로써 I/O 비용을 줄이는데 사용된다.

Haloop에서 Cache를 이용할 때 만약, Cache를 로드할 수 없는 상황이라면, Reducer Input Cache는 mapper의 첫 반복에서 나온 결과를 다시 가져오며, Reducer Output Cache와 Mapper Input Cache는 HDFS나 local file system에서 데이터를 가져온다.

○ **analysis of experimental results**

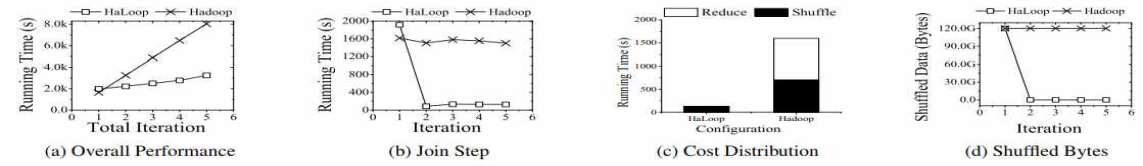

그림 4 Evaluation of Reducer Input Cache

그림 4는 1개의 마스터 노드와 50개의 슬레이브 노드로 구성된 컴퓨터 환경에서 Livejournal Dataset을 Page Rank 알고리즘을 이용하여 분석하였을 때 Hadoop과 Reducer Input Cache를 사용한 Haloop의 성능을 비교한 것이다. (a) Overall Performance를 보면 반복이 진행될수록 Haloop의 성능이 Hadoop보다 좋아지는 것을 확인할 수 있다. (b)Join Step에서는 첫 번째 반복에서 Haloop이 Hadoop보다 성능이 안 좋고, 이후 반복에서는 좋은 것을 확인할 수 있는데, 이는 Haloop에서 첫 번째 반복에서 Cache를 만드는 추가적인 작업이필요하기 때문이다. 이후 반복에서는 Haloop은 reducer input cache를 사용하기 때문에 Hadoop보다 빠른 성능을 보인다. (c) Cost Distributuion에서는 Hadoop과 비교해서 Haloop이 Reduce time과 Shuffle time을 합한 결과가 더 적은 것을 확인할 수 있는데 이는 Haloop에서는 Cache를 저장하고 있어서 Shuffle TIme과 Reducer로 모인 데이터를 정렬하고 그룹핑하는 Reduce Time을 줄일 수 있기 때문이다. 마지막으로 (d) Shuffled Bytes를 보면 1번째 반복에서는 동일한 데이터를 섞고, Hadoop은 Shuffle Data의 차이가 발생하지 않는 반면, Haloop은 크게 감소한 것을 볼 수 있는데 이는 해당 데이터를 Cache로 각각의 Reducer에 저장해 놓는 Haloop의 특성 때문이다.

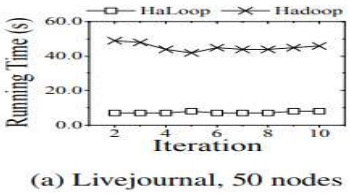

그림5 Evaluation of Reducer Output Cache

그림5는 Haloop에서 Reducer Output Cache를 사용하였을 때, Reducer Input Cache와 동일환 환경과 알고리즘을 이용하여 Livejournal DataSet을 분석 하였을 때 Hadoop과의 성능을 비교한 것이다. 해당 그림에서 Haloop이 fixpoint를 평가할 때 Hadoop보다 성능이 좋은 것을 알 수 있다. Hadoop은 각각의 fixpoint의 결과를 재 취합하는 맵리듀스 잡이 추가로 필요하기 때문이다.

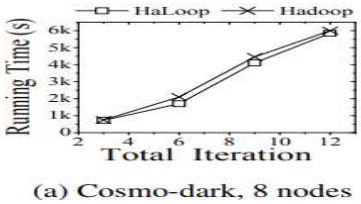

그림 6 Evaluation of Mapper Input Cache

그림6은 Haloop에서 Mapper Input Cache를 사용하였을 때, 8개의 노드를 사용하는 환경에서 Cosmo-dark 데이터를 k-means 알고리즘을 이용하여 분석하였을 때 Hadoop과의 성능을 비교한 것이다. Haloop에서 Mapper Input Cahce를 이용하여 Hadoop보다 미세하게 성능이 좋아진 것을 알 수 있다.

○ **opinion or improvement point**

Haloop의 Inter-iteration locality와 L에 대한 Cache를 유지하는 방법은 반복 프로그래밍에서 I/O 비용을 감소시키기 위한 효과적인 방안이라고 생각한다. 그러나 논문을 읽으면서 아쉬웠던 점은 Haloop에서 기본적인 API만 제공한다는 점이였다. Haloop에는 3가지의 Cache가 존재하고 각각의 Cache마다 효과적인 알고리즘들이 존재한다. 때문에 사용자가 Haloop을 이용하여 빠른 성능을 얻기 위해서는 자신이 사용하는 알고리즘에 대한 깊은 이해와 해당 알고리즘이 Haloop에서 제공하는 Cache 중에서 어떤 것을 사용해야 하는지 알아야하며, 이들을 여러 가지 저수준 API를 이용하여 설정해야한다. 그러므로 대표적인 알고리즘들에 한해서 각각 어떤 Cache를 사용해야하는지를 Haloop에서 만들어서 Document로 제공해주거나, 사용자가 간단한 입력값으로 반복 데이터를 처리할 수 있도록 High Level APPlication을 제공해줬다면 사용자들이 Haloop을 좀 더 쉽게 사용할 수 있도록 할 수 있을거라 생각한다.