MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
Google, Inc

2015104221 정준현

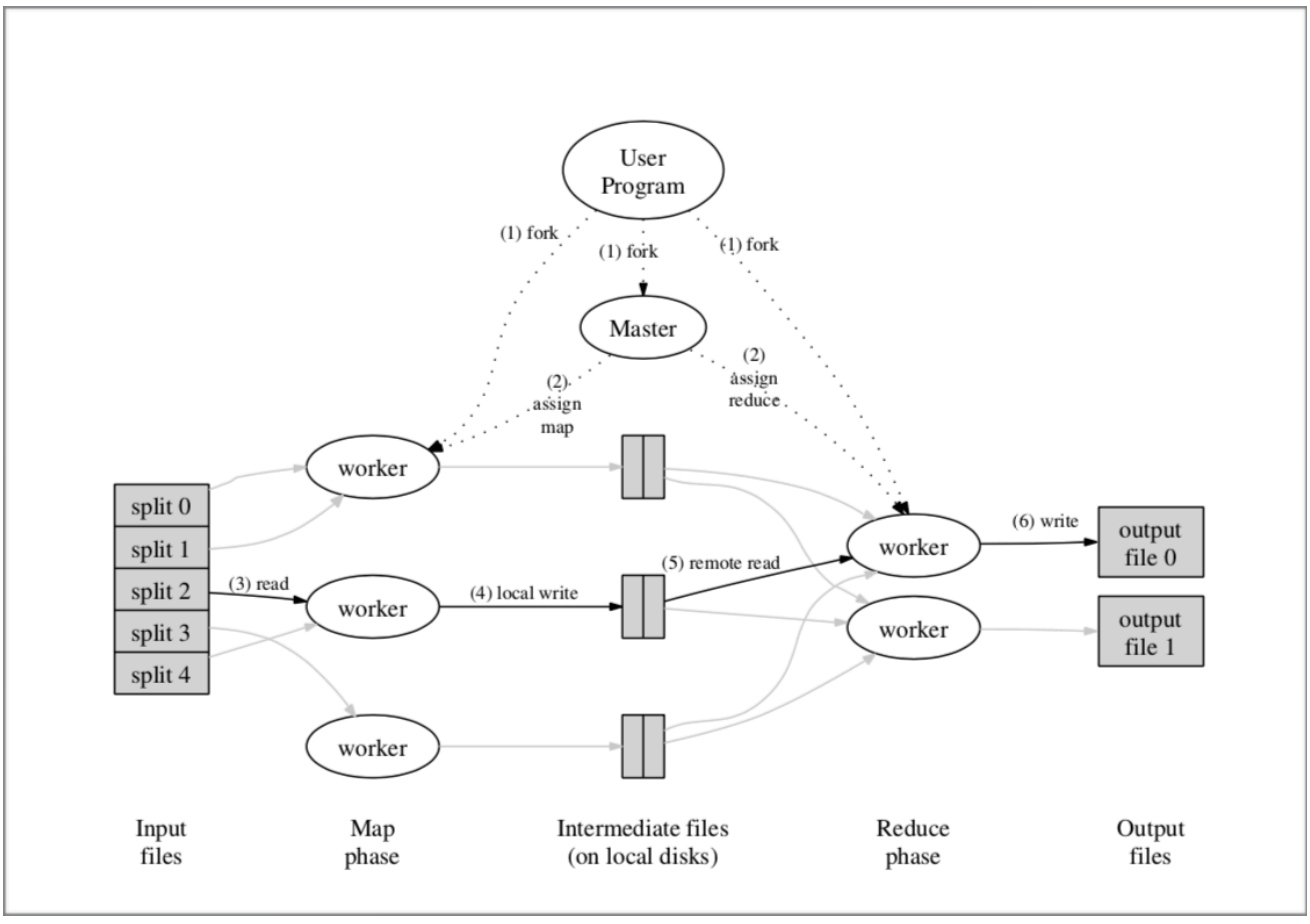
1. Motivation

2004년 당시의 구글에서는 많은 양의 raw data를 분산된 computation 환경을 이용해서 일정 시간 내에 처리해야 했습니다. 이를 이루기 위해 어떻게 computation을 병렬화해야 하는지, input data를 어떻게 분산시켜야 하는지, machine failure에 대해서는 어떻게 처리해야 하는지에 대한 이슈를 해결해야 했습니다. 또한, 대부분의 구글의 computation 과정이. map reduce를 적용하고 있는 것을 발견하였습니다.

이러한 issue와 observation을 바탕으로 Jeffrey Dean과 Sanjay Ghemawat는 사용자가 map과 reduce operation을 직접 작성하도록 하는 model을 구현하였다고 합니다. 즉, 사용자는 computation 병렬화, fault tolerance, input data. distribution 과정은 구현 하지 않아도 되며 map reduce operation에만 집중할 수 있도록 하였다고 합니다.

2. Method or solution

2.1 Map Reduce Execution



<Figure 1. Map Reduce Execution Overview>

Figure 1는 Map Reduce operation의 과정을 나타낸 그림입니다. 그림의 일련의 과정은 다음과 같이 설명할 수 있습니다.

(1) fork: 먼저 User program이 Map reduce function을 호출하게 되면, input file을 16~64 Mbytes 단위의 piece로 쪼개게 됩니다. 이어서 많은 program의 복사가 일어나게 됩니다.
(2) assign map & reduce: 복사된 program은 크게 Master와 Worker로 분류됩니다. Master는 Worker에게 적절하게 스 map 혹은 reduce task를 scheduling 해줍니다.
(3) Read and local write: Map task가 부여된 Worker는 input split에 대해서 key, value를 parse 하여 map function에 던져주게 됩니다. map function으로부터 나온 intermediate key value는 메모리에 buffer 되었다가, local disk에 쓰이게 됩니다. 이 때, 이 local disk의 buffered pair의 위치가 master로 전달되게 되고, 후에 master는 Reduce worker에게 이 위치를 전달하게 됩니다.
(4) Remote read and write: Master는 intermediate key value pair들이 local disk의 어떤 위치에 쓰였는지 알고 있으므로, Reduce worker는 이 위치를 RPC(Remote Procedure Call)를 통해 읽게 됩니다. 이 때, intermediate key 값들에 대해서 정렬하여, 같은 intermediate key는 서로 묶여질 수 있도록 합니다. 그렇게 해서 묶인 모든 유일한 Intermediate key와 value pair를 reduce function에 반복적으로 전달해줍니다. Reduce function에서 나온 output이 최종적으로 output file에 쓰이게 됩니다.
(5) Finish: Map task와 Reduce task가 모두 끝나게 되면, Master는 user program를 wake up하여 Map Reduce operation이 종료되게 됩니다.

2.2 Failure tolerance

Worker failure tolerance를 보장하기 위해서, master는 주기적으로 모든 worker에 대해서 핑을 보냅니다. 일정 시간 동안 이에 대한 반응을 보내지 않으면 master는 해당 worker와 task를 fail로 표시합니다. 모든 실패한 task와 worker는 master에 의해 idle state로 돌아가게 됩니다. map task는 output이 local disk에 씌여지게 되는데, map task를 소유하고 있던 worker machine이 고장 나게 되면 local disk에 접근할 수 없어 output file을 사용할 수 없으므로 다른 worker machine에서 재실행되어야 합니다. 반면, reduce worker의 경우 output이 global file system에 쓰여지기 때문에 fail 되어도 재실행될 필요가 없습니다.

Master failure tolerance를 보장하기 위해서, 주기적으로 master의 data structures(worker, task들의 state를 관리하는 데이터 구조)를 복사하여 checkpoint를 만듭니다. 그리하여 master worker machine이 fail 난다면, 최근의 checkpoint로 master를 재 시작할 수 있습니다. 그러나 master가 한 개인 경우에는 위와 같은 해결법으로는 해결할 수 없는 상황(예를 들어, master의 모든 checkpoint를 손실한 경우)이 존재합니다. 이러한 경우에는 Map Reduce computation을 중지하고 Clients가 Map Reduce Operation을 다시 시작하게끔 하였습니다.

2.3 Locality

Network 대역폭은 computing 환경에서 아주 희소한 자원입니다. Network 대역폭을 줄이기 위해서, 본 논문에서는 Locality Optimization mechanism을 도입하였습니다.

GFS(Google File System)에서는 Input data block을 64MB로 쪼개고, 각 Input data block에 대해서 보통 3개의 복제본을 다른 machine에 저장하게 됩니다. 이 때, Locality Optimization이란 Master는 최대한 input data block 원본을 가지고 있거나, 복제본을 가지고 있는 machine에 map task를 부여하도록 하는 정책을 의미합니다. Map task에서 각기 다른 여러 개의 machine에서 data block을 읽게 되면 network bandwidth가 커지기 때문에 위와 같은 mechanism을 도입하였다고 합니다.

2.4. Backup tasks

map reduce operation을 진행하면서, 비정상적으로 오래 걸리는 task들이 존재하게 되는데, 이를 straggler라고 합니다. Straggler가 생기는 원인은 다양합니다. 예를 들어, Bad disk를 가진 machine이 Read Performance를 30MB/s 에서 1MB/s로 늦추는 에러를 지속적으로 만나는 경우, Cluster Scheduling System이 Map Reduce Operation 말고도 다른 작업들을 스케쥴링 하여 CPU, local disk, network bandwidth 에 대한 경쟁이 생기는 경우에는 Stragglers가 발생할 수 있습니다 . 이러한 straggler들이 전체 수행 시간을 길게 하므로, map reduce operation의 막바지쯤 master는 남은 작업들에 대해서 backup execution을 다른 idle 상태인 worker에게 스케줄링 합니다. 하여 backup과 straggler 중 먼저 작업이 끝난 task를 master는 completed task라고 표시하게 되고, straggler로 인한 전체 수행 시간이 늘어나는 현상을 어느정도 해결해줄 수 있게 됩니다. 본 논문에서는 BackUp task mechanism을 tuning하여 Computing resource 사용률을 몇퍼센트 안될정도로 적게 올릴 수 있었다고 합니다. 또한, Backup task mechanism을 이용하여, sorting program에서 전체 수행시간이 약 44퍼센트 빨라졌다고 합니다.

2.5 Partitioning function

R을 사용자가 원하는 reduce task 와 output file의 개수라고 가정하겠습니다.
Reduce function의 결과로 Output data는 partitioning function에 의해서 분할되어 output file에 쓰여지게 됩니다. 이때 보통의 partitioning function은 Eq 1. 과 같습니다. 그러나 특별한 경우 (예를 들어, 같은 hostname을 가지는 urlKey에 대해서 같은 output file에 넣고 싶은 경우)에는 partitioning function을 Eq 2. 와 같이 설계할 수 있습니다.

$$hash(key)\ mod\ R$$

<Eq 1. Default partitioning function>

$$hash(Hostname(urlKey))\ mod\ R$$

<Eq 2. Customized partitioning function>

2.6 Ordering Guarantees

본 논문에서는, Intermediate key 와 value pair 에 대해서 key값으로 pair가 increasing - order로 정렬되는 정책을 채택하였습니다. 그 이유는 이러한 정렬이 partition 마다 정렬된 output file을 만들어내기가 쉽기 때문입니다. 즉, key를 가지고 Random Access lookups를 할 경우나 사용자가 output data가 정렬 되기를 바라는 요구 사항을 충족하는데 유용하게 쓰일 수 있습니다.

2.7 Combiner function

Network를 통해 Reduce function으로 수백 수천 개의 intermediate key value pair가 들어가기 전에, 미리 partial merging을 해줄 수 있는 Combiner function을 유저로 하여금 선택적으로 작성할 수 있게 하였습니다. Reduce Wokrer의 부담을 줄여주기 위해서, 이 작업 이전에 combiner function은 reduce function이 해줄 일을 미리 해주게 됩니다. Combiner function은 map task를 실행하고 있는 machine에서 실행되며, Combiner function의 output은 intermediate file에 쓰여지게 됩니다. 이 때 Combiner function의 코드와 Reduce function의 코드는 동일하다고 합니다. 결과적으로, Combiner function을 사용하여 속도를 굉장히 높일 수 있다고 합니다.

2.8 Skipping Bad Records

유저가 작성한 Map Reduce function에는 특정 Record에서 오류가 나는 코드가 있을 수 있습니다. 이러한 버그들은 Map Reduce operation이 끝나지 않게 합니다. 이러한 Bug에 대해서 먼저 고치는 것이 우선 순위 이기는 하나, 에러를 내는 코드가 Third-Party library일 경우와 같이 Bug를 해결할 수 없는 상황이 있을 수 있습니다. 따라서 이러한 일부의 Bad Records에 대해서 Map Reduce Library가 Crash를 감지하게 되면, 해당 records를 선택적으로 skip할 수 있는 정책이 Skipping Bad Records 입니다.
각각의 Worker는 Signal Handler를 둬서 Segmentation과 Bus error를 감지하면 Signal을 보내게끔 합니다. Map Reduce Operation이 시작되기 전에, Map Reduce library는 argument의 sequence number를 전역 변수로 저장합니다. User code가 signal을 보내면, Signal Handler는 sequence number가 들어있는 “last gasp” UDP packet을 Master에게 보내게 됩니다. Master에서 해당 레코드에 대해서 Fail이 났다는 UDP packet을 한번 이상 받게 되면, 다음번에 Map Reduce 재 실행시 해당 레코드를 Skip하게 됩니다.

2.9 Counters

Counters는 Map reduce operation이 진행되면서 여러가지 발생되는 이벤트에 대해서 count 하기 위한 것입니다. 사용자가 중간중간에 원하는 값을 보고 싶을 때 counters를 이용할 수 있습니다.
Counters가 동작하는 방식은 다음과 같습니다. 각 Worker machine의 Counter 값들은 주기적으로 Master에게 전달됩니다. 이후 Master는 성공적으로 마친 map, reduce task로 부터 Counter 값들을 모은 후, Map Reduce operation이 끝날 때 user code에게 돌려주게 됩니다. Master Status Page에 현재 Counter 값들이 보여지므로, 사용자는 computationon의 진행을 실시간으로 확인할 수 있게 됩니다.
Counters는 Sanity checking(온전성 검사)에도 유용하게 쓰일 수 있습니다. Input pair의 개수와 최종 output pair의 개수등 몇 가지 Counter 값들은 Map Reduce Library에 의해서 자동적으로 관리되어집니다. 이 때, Counters를 이용하여 사용자가 최종 output pair의 개수와 input pair의 개수가 동일한 지 확인하는 등의 map reduce operation 온전성 검사를 진행할 수 있습니다.

3. Analysis of experimental results

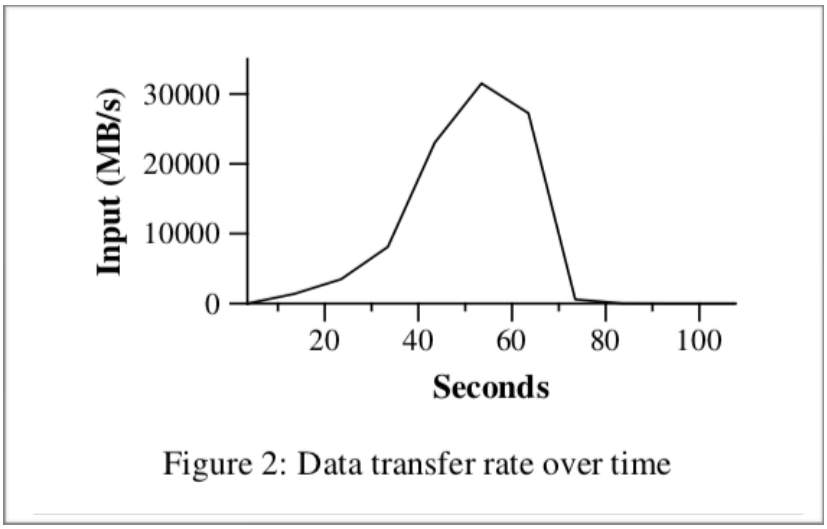


Figure 2: Data transfer rate over time

<Figure 2. Performance on Grep>

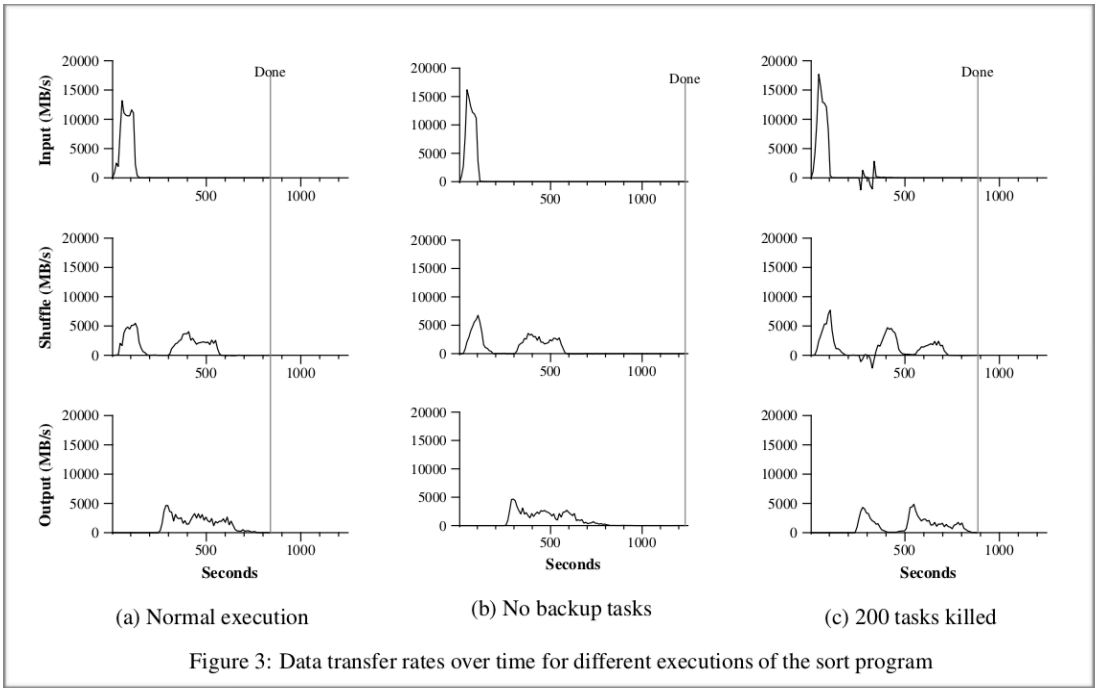


Figure 3: Data transfer rates over time for different executions of the sort program

<Figure 3. Performance on Sorting program>

Figure 2는 1테라바이트의 데이터중 특정 패턴을 찾는 프로그램에 대해서 map reduce performance를 측정한 그림입니다. 이 그래프에서, 시간이 지나면서 더 많은 machine의 worker가 할당되면서 약 1780개의 worker가 할당되었을 때 30GB/s의 최고 속도로 input data를 스캔하였습니다. 총 computation 시간은 여러 가지 overhead를 포함하여 약 150초 정도 걸렸습니다.

Figure 3는 1테라바이트의 데이터를 sorting하는 프로그램에 대해서 map reduce performance를 측정한 그림입니다. 이 그래프에서 맨왼쪽 위 그림의 input data 스캔하는 속도의 최고점이 13GB/s 정도로 grep에 비해서 낮은 이유는 ,

intermediate output을 local disk에 쓰는 작업이 많아 이에 대한 I/O 점유율이 높아져 input data를 스캔하는 속도는 상대적으로 느려지게 됩니다.
또한 normal execution했을 때의 총 걸린 시간은 891초, backup task를 적용하지 않았을 때는 1283초 정도로 normal execution에 비해 44%정도 느리다고 합니다. 또한 그래프 (c)에서, 1700개의 worker중 일부러 200개의 worker를 죽여도 전체 걸린 시간은 normal execution에 비해 5% 정도로 미세하게 늘었다고 합니다. 이는 Worker tolerance 정책 덕분이라고 추론할 수 있습니다.

4. opinion or improvement point

- Map Reduce 모델은 컴퓨팅 병렬화, fault tolerance, locality 최적화 등을 보장하는 사용하기 쉬운 프로그래밍 모델입니다.
- 이를 통해 많은 도메인의 문제를 해결 할 수 있습니다.
- 많은 컴퓨팅 자원을 효율적으로 이용할 수 있습니다.

그러나, 본 논문에서 제시한 mechanism에서 분명한 Improvement Point가 존재합니다.
첫째로, Map Reduce operation 초기 단계에서 Master가 Worker에게 미리 map task 혹은 reduce task를 정해주는 방식을 개선해야 된다고 생각합니다. CPU 사용량, 작업 진행률, I/O 처리 점유율 등을 고려하여 동적으로 Map task혹은 Reduce task를 정해주지 않고 미리 정해버리면, 효율적으로 Map Reduce Computation을 처리 할 수 없습니다.

두번쨔로, Master가 fail날 경우에 대한 확고한 정책이 필요하다고 생각합니다. 본 논문에서는 Master가 fail나면 가장 최신의 checkpoint로 부터 Master를 재시작하게 되는데, Checkpoint를 손실했다던지의 문제가 생긴다면 Master를 복구 할 수 없고, 오로지 Map Reduce Operation을 재시작하는 방안을 해결책으로 제시하였습니다. Checkpoint를 손실한 경우와 같이 Master를 복구할 수 없는 경우에 대한 해결책이 필요하다고 생각합니다.