



Week 4

<Question 1>

(Week 3 과제 구상도 입력)

이 구조는 shuffling 과정에서 병렬화 되어있지 않은 문제가 있다. 어떻게 바꾸는 것이 좋을까?

<Answer 1>

Shuffling 과정의 병렬화를 고려한다면, 각 Worker가 독립적으로 병렬로 데이터를 전송하고 받는 방식을 적용할 수 있습니다.

기본 아이디어는 Worker들이 병렬로 자신의 데이터를 준비하고, 필요에 따라 동시에 데이터를 교환할 수 있게 하는 것입니다.

각 Worker가 순서에 따라 데이터를 순차적으로 전달하기보다는 가능한 모든 Worker가 서로 동시에 통신하고 필요한 데이터를 주고받는 형태로 수정할 수 있습니다.

여기서 병렬화 방안을 위해 적용할 수 있는 몇 가지 방법을 제안합니다:

1. Worker 간의 비동기 데이터 전송

기존 방식은 Master가 Worker를 지정하여 순서대로 데이터를 교환하므로, 한 Worker의 작업이 끝나야 다음 Worker가 진행됩니다.

이 방식을 비동기 통신 방식으로 변경하면 각 Worker가 병렬로 데이터를 전송 및 수신할 수 있습니다.

각 Worker는 독립적으로 자신이 보낼 데이터 블록을 준비하고, 다른 Worker에게 비동기적으로 데이터를 전송합니다.

데이터를 받는 Worker는 수신한 데이터 블록을 큐에 삽입하여 순서대로 정렬하고, 다른 Worker가 데이터를 더 보내기를 기다리지 않고 정렬 작업을 이어갈 수 있습니다.

이 방식에서는 Master가 특정 Worker를 선택하여 집중적으로 데이터를 전송하는 대신, 모든 Worker가 필요할 때마다 비동기적으로 데이터를 보내도록 구성합니다.

2. 각 Worker가 Shuffling 역할을 분담

Shuffling 과정을 Worker 간에 분산하여 실행할 수 있습니다. 기존에는 Master가 Worker를 지정하는 방식으로 모든 작업이 직렬화되었지만, 다음과 같이 변경할 수 있습니다.

각 Worker는 자신이 담당할 Partition을 정하고, 해당 Partition을 완성하는 데 필요한 데이터를 각 Worker로부터 받습니다.

Worker 간의 데이터 전송이 동시다발적으로 이루어지며, 모든 Worker는 데이터 수신과 정렬 작업을 독립적으로 수행합니다.

Master는 각 Worker의 상태를 추적하는 역할만 수행하고, 특정 Worker에 집중적으로 데이터를 전송하는 방식에서 벗어납니다.

3. 데이터 교환과 정렬을 위한 피어-투-피어 방식 적용

Worker들 간에 피어-투-피어 통신 방식을 적용하여 Shuffling 단계의 병목을 줄일 수 있습니다.

Worker들이 서로 필요한 데이터만 교환하고, 전체 데이터의 이동이 끝나지 않아도 일부 Worker는 Merge Sort를 진행할 수 있도록 합니다.

예를 들어, 각 Worker가 자신이 필요한 데이터 블록을 다른 Worker로부터 수집한 후, 일부 데이터가 도착했을 때 바로 병렬 정렬을 시작할 수 있도록 합니다.

4. 데이터 전송을 위한 Multi-threading 사용

각 Worker가 자체적으로 여러 스레드를 사용하여 데이터를 전송하고 받는 작업을 동시에 수행할 수 있습니다.

Worker는 여러 스레드를 활용해 데이터를 준비하고, 각 스레드가 다른 Worker와 병렬로 통신하여 필요한 데이터를 교환합니다.

데이터가 준비되면 각 Worker는 바로 정렬을 시작할 수 있고, 추가 데이터가 들어오면 이어서 병합하여 최종 정렬을 수행합니다.

이러한 방식으로 Shuffling을 병렬화하면, 데이터 전송과 정렬의 대기 시간이 줄어들고 전체 프로세스의 성능이 개선될 수 있습니다.

병렬 통신과 Worker 간 데이터 교환의 비동기화가 핵심이며, 이를 통해 기존의 직렬적인 병목을 해소할 수 있습니다.

<Idea 1>

기존 코드에서 shuffling을 순차적으로 실행한 이유는 다음과 같다.

1. disk에 남은 공간이 부족할 수 있기에 각 worker들이 어디로 data를 보내야 할 지 복잡해진다
2. Input과 Output을 동시에 진행할 수 없다고 생각했다.

gpt의 제안 4번에서 얻은 아이디어로, Concurrent programming을 활용하면 2번 문제가 해결되어 I/O를 동시에 활용할 수 있게 된다.

1번 문제를 해결하기 위해 새로운 메시지를 도입하는 것을 고려해보았다.

각 worker는 disk 용량이 전부 차 있을 시 <data 받지 않음> 메시지를 전송한다.

용량이 부족한 worker에게 데이터를 보내려고 시도한 후, <data 받지 않음> 메시지를 받으면 다른 부분 부터 전송하도록 한다.

이 때, worker 사이에 메시지를 주고 받도록 구현하면 경로가 최대 $n(n-1)$ 개 (complete directed graph) 로 매우 많아 복잡해진다.

따라서 메시지는 worker -> master -> worker 의 방식으로 구현해야 경로 $2n$ 개로 줄어들어 복잡성이 줄어든다.

따라서, 세부적인 구상은 다음과 같다.

<data 전송>

- 각 worker는 random하게 보낼 data를 정한다.
- data를 정할 때 어느 worker에 보낼 지 정하는 것이 좋아 보인다.
- random 선택할 때 자신의 data는 보내지 않도록 한다. (queue의 앞부분만 탐색)
- 해당 worker의 정보를 master에게 요청한다.
- master는 해당 worker가 free한지 full인지 정보를 보낸다.
- 또한 master에게 해당 worker의 남은 용량이 얼마인지 전달 받는 것이 더 좋을 것 같다.
- 해당 worker가 full 이면 다른 key range의 data를 탐색하여 master에 다시 요청한다
- 해당 worker가 free 이면 worker에게 data를 전송한다.
- 보낼 data의 크기 > data를 받을 worker의 용량 이라면 data를 남은 용량 크기만큼 보낸다.
- data를 보낸 후 master에게 자신의 남은 용량을 갱신하는 메시지를 보낸다.

<data 수신>

- data를 전달 받으면 자신의 disk 용량이 full인지 master에게 메시지를 보내 갱신한다.
- data를 받은 후 master에게 자신의 남은 용량을 갱신하는 메시지를 보낸다.
- 받은 data는 queue의 뒷부분에 차례대로 삽입한다.

이 과정에서 발생하는 문제점은 다음과 같다.

1.

queue는 FIFO 구조로서, 앞부분 부터 꺼낼 수 있다.

이렇게 되면 key range가 가장 앞부분인 worker에게 집중적으로 data를 전송하게 되어 기존의 구현과 유사해진다.

따라서 data를 전송할 대상을 random하게 정하거나 특정 알고리즘을 통해 정해야 하는데, FIFO

구조가 이를 방해한다.

2.

만약 둘 이상의 worker에게서 data를 전달받을 때 용량을 초과하는 문제가 발생할 수 있다.
이 문제를 해결할 방법으로 master에게 data를 보낼 대상을 확인하는 메시지를 보내는 과정을 수정하는 방법을 사용할 수 있다.

<Idea 2>

worker B -> worker A 로 data를 전송하는 과정에서

master는 B에게 A의 data 제공을 요청받는다. 이 때 master는 A를 writing flag를 on 시켜 다른 worker가 A에 data를 전송하지 못하도록 한다

worker C -> worker A 로 data를 또다시 전송한다 하면

master는 먼저 A의 writing flag를 확인한다.

이 때 flag on 이면 C에게 반려 메시지를 보내고, flag off면 data 전송을 진행하는 메시지와 A의 정보를 보내고, flag를 on 시킨다.

이 방법을 사용하면 동시에 writing되는 문제는 해결할 수 있으나, 불필요한 작업을 반복할 가능성이 높다.

이 flag들을 이용해서 data를 수신할 worker를 정하는 알고리즘에 도움을 줄 수 있을 것 같다.

먼저 worker는 master에게 writing flag를 요청하여 메시지를 받고, writing flag off 인 worker를 선택하여 메시지를 보내는 것이 좋을 것 같다.

<Question 2>

(위의 <Idea 1> 부터 <Idea 2> 끝까지 전부 복사 후 입력)

Idea 2 의 correctness는 어떤가? queue 구조를 해결 할 방법이 뭘까?

<Answer 2>

Idea 2와 관련해 correctness를 보장하고 queue 자료구조를 개선하는 방안을 아래와 같이 제안합니다.

1. Idea 2의 Correctness 보장

Idea 2에서는 writing flag를 사용하여 동시에 여러 Worker가 동일한 대상에게 데이터를 보내는 문제를 방지하고 있습니다.

이 접근 방식은 통신을 조정하여 데이터 전송을 안정적으로 수행할 수 있게 하지만,

몇 가지 추가적인 보완 사항이 필요합니다.

Master의 Writing Flag 관리:

각 Worker가 특정 Worker로 데이터를 전송하기 전에, Master에게 해당 Worker의 writing flag를 확인해야 합니다.

이때 Master가 실시간으로 writing flag 상태를 정확하게 관리하여 flag 상태를 효율적으로 업데이트하고 요청을 차례대로 처리해야 합니다.

Queue Overflow 방지:

각 Worker가 여러 데이터 요청을 받을 수 있기에, Master가 Worker의 writing flag 상태와 남은 용량 정보를 동시에 관리하여 각 Worker의 용량을 초과하지 않도록 해야 합니다.

이는 Worker들이 Master에게 데이터를 보낼 준비 상태와 용량 정보를 자주 업데이트하도록 하여 정확한 상태를 반영할 수 있게 하는 것이 중요합니다.

Data Transfer Retries:

Worker가 데이터를 전송하려 할 때 대상 Worker의 용량이 부족한 경우 또는 writing flag가 계속 켜져 있는 경우에는 Master가 대기열에 해당 요청을 보류하고, 일정 시간이 지나거나 writing flag가 꺼질 때 다시 전송을 시도할 수 있는 재시도 로직을 추가하는 것이 좋습니다.

이를 통해 Worker 간의 비효율적인 재시도 및 중복 전송을 방지할 수 있습니다.

2. Queue 구조 개선

Idea 2에서는 FIFO queue 구조가 특정 Worker에만 데이터를 집중적으로 전송하게 되는 문제를 언급했습니다.

이를 개선하기 위해 queue 구조를 적절히 변경하여 데이터를 분산시키고 전송 대상을 효율적으로 관리할 수 있습니다.

a. Multi-Level Queue 활용

FIFO queue를 여러 레벨로 나누는 Multi-Level Queue 방식을 고려할 수 있습니다.

각 key range에 대한 별도 queue 생성:

각 Worker가 보내야 할 key range별로 개별 queue를 만들어, 모든 key range에 대한 queue가 있는 상태로 유지합니다.

랜덤 순서 또는 Round-Robin 방식으로 queue 선택:

데이터를 전송할 때는 특정 Worker의 queue를 앞부분부터 탐색하는 대신, 여러 queue 중 하나를 선택하여 queue 간의 부하를 분산시킵니다.

Round-Robin 또는 랜덤 선택을 통해 전송 대상을 분산시키면 특정 Worker에 집중되는 문제를

완화할 수 있습니다.

Master에서 queue 우선순위 조정:

Master가 queue를 순차적으로 탐색하며 가장 작은 부하를 가진 queue부터 데이터 전송을 시작하도록 설계하여 효율성을 높일 수 있습니다.

b. Priority Queue 도입

특정 기준에 따라 우선순위를 결정하여 queue에 넣고, 우선순위가 높은 대상을 먼저 처리하도록 합니다.

Worker의 용량에 따른 우선순위: 남은 용량이 많은 Worker가 우선적으로 데이터를 받을 수 있게 하여 데이터 집중을 방지합니다.

writing flag 상태에 따라 우선순위 변경: writing flag가 꺼져 있는 Worker에 우선순위를 높여 데이터를 전달하며, 다른 Worker의 대기 시간이 짧아질 수 있게 합니다.

특정 조건에서 랜덤화 추가: 특정 대기 시간이 경과하거나, 여러 Worker의 용량이 유사할 때 랜덤하게 Worker를 선택하여 전송을 수행하도록 합니다.

결론

Idea 2의 correctness를 보장하려면 Master가 writing flag와 Worker의 용량을 동적으로 관리하여 적절한 Worker에게 데이터를 전송할 수 있도록 하고,

이를 위해 queue를 개선하여 Worker 간 데이터 집중을 방지하는 것이 중요합니다.

Multi-Level Queue 또는 Priority Queue를 활용하면 데이터 전송 대상을 분산하여 성능을 최적화할 수 있습니다.

<Idea 3>

priority queue를 사용하는 편이 굉장히 좋아보인다.

우선 순위를 worker의 용량에 따라 정한다고 하였는데, (worker에게 보내야 할 data 크기) 값 또한 우선 순위 결정에 중요한 역할을 한다.

$(\text{worker 용량}) - (\text{worker에게 보내야 할 data 크기}) > 0$ 인 경우라면 한 번의 data 전송으로 더 이상 해당 key range를 고려하지 않아도 되어 좋다.

문제는 (worker 용량)과 (worker에게 보내야 할 data 크기) 둘 다 클 수록 좋다는 것이다.

data 전송량이 클 수록 빈 공간이 많아지므로 더 많은 data를 주고받을 수 있어 좋다.

data 전송량을 높이기 위해서는 worker의 용량이 가장 우선적으로 고려되어야 하며, worker에게 보내야 할 data 크기는 부가적이다.

다만 worker에게 보내야 할 크기가 큰 것도 중요하므로, 두 값으로 priority를 계산하는 식을 결정

하는 것이 중요해 보인다.

<Question 3>

(<Idea 3> 전체)

priority 계산 식을 어떻게 정하면 좋을까?

<Answer 3>

priority 계산식에서, Worker의 남은 용량과 보내야 할 데이터 크기를 모두 고려하여 각 Worker의 우선순위를 정하는 방식으로 구성하면 됩니다.

여기서 중요한 점은 두 요소가 함께 작용해야 하지만, 남은 용량이 우선적으로 중요하다는 것입니다.

이를 반영한 priority 계산식의 몇 가지 접근 방법을 아래에 제안합니다.

1. Weighted Linear Combination (가중치를 부여한 선형 조합)

우선순위(priority)를 Worker의 남은 용량과 보내야 할 데이터 크기에 서로 다른 가중치를 부여하여 계산합니다.

$$\text{priority} = w_1 \times (\text{남은 용량}) + w_2 \times (\text{보낼 데이터 크기})$$

여기서 w_1 과 w_2 는 각각 남은 용량과 보낼 데이터 크기의 가중치를 의미합니다. 예를 들어, $w_1 = 0.7$ 및 $w_2 = 0.3$ 과 같이 설정할 수 있습니다.

2. Capacity Utilization Ratio (용량 활용 비율)

남은 용량 대비 보낼 데이터 크기의 비율을 구하여 우선순위를 정하는 방식입니다.

$$\text{priority} = \frac{\text{보낼 데이터 크기}}{\text{남은 용량}}$$

이 방식에서는 보낼 데이터 크기가 남은 용량과 비슷할수록 우선순위가 높아지는 효과가 있습니다.

이 방식을 적용할 때 1에 가까운 값이 가장 효율적이므로, priority queue에서는 우선순위를 (1 - priority)로 설정하여 가장 낮은 값을 우선으로 정렬할 수 있습니다.

3. Multiplicative Balance (곱을 이용한 균형 계산)

남은 용량과 보낼 데이터 크기를 곱하여, 두 값이 모두 클 때 우선순위를 높이는 방식입니다.

$$\text{priority} = (\text{남은 용량}) \times (\text{보낼 데이터 크기})$$

이 방법은 남은 용량이 큰 Worker에 큰 데이터 전송을 우선시하게 합니다.

4. Sigmoid Function을 사용한 스무딩

남은 용량과 보낼 데이터 크기 사이의 균형을 부드럽게 조절하고자 할 때 Sigmoid 함수를 사용할 수 있습니다.

$$\text{priority} = \sigma(\alpha \times \text{남은 용량}) + \beta \times \sigma(\gamma \times \text{보낼 데이터 크기})$$

여기서 $\sigma(x) = \frac{1}{1+e^{-x}}$ 로 정의되며, 남은 용량과 보낼 데이터 크기가 특정 임계치를 넘었을 때 급격히 우선순위가 높아지게 조정할 수 있습니다.

결론

각 방법의 특징을 요약하자면:

Weighted Linear Combination: 가중치를 조정하여 단순하게 적용하기 좋음.

Capacity Utilization Ratio: 남은 용량 대비 데이터를 효율적으로 보낼 수 있는 Worker를 우선시할 때 유리.

Multiplicative Balance: 두 값이 모두 클 때 우선순위를 높여 큰 데이터를 큰 용량에 전송.

Sigmoid Function: 특정 임계값을 넘기면 우선순위를 급격히 높이도록 조정 가능.

가장 적합한 방법은 시스템의 요구사항에 따라 다르겠지만,

Weighted Linear Combination과 Capacity Utilization Ratio가 직관적이고 간단하게 적용하기 좋을 것입니다.

2 번은 (worker 남은 용량) 과 (보낼 data 크기) 가 반비례 하므로 좋은 선택이 아닌 것 같아 보인다. 두 값 모두 커질 때 priority가 커지는 편이 좋을 것 같아 보인다. 3 번은 (worker 남은 용량)을 우선적으로 고려하는 요소가 없어서 좋은 선택이 아닌 것 같아 보인다. 4 번은 조금 더 계산을 해 보아야 하겠지만 1 번을 선택하는 것이 가장 좋아 보여 고려하지 않으려고 한다. 위 3가지 방법 모두 project 구현이 끝나면 대체할 수 있는 요소로서 생각해 보는 것으로 한다.

따라서 priority 계산은 linear combination을 이용하는 편이 좋을 것 같다. parameter는 구현 도중 적절한 값을 찾는 것으로 한다.