프로젝트 최종보고서

네임노드 최적화를 통한 마스터 메모리 절감

텍스트, 엠블럼, 로고, 원이(가) 표시된 사진

자동 생성된 설명]

**팀: 6조 이석진, 문무현**

**제출 일시: 2024 - 06 - 22**

목차

1. 팀원 소개

2. 작품 소개

2.1 핵심 문제

2.2 해결 방법

2.3 SW 설계

2.3.1 기능 요구사항

2.3.2 비기능 요구사항

2.3.3 SW 설계도

2.3.4 최종 산출물 구성

3. 예상되는 문제

3.1 Risk Analysis

3.2 Risk Reduction Plan

4. 코드 소개

4.1 서버 코드 소개

4.2 클라이언트 코드 소개

5. 테스트 방법

5.1 입력 파일

5.2 출력 파일

6. 수행 결과

7. 참고 자료

**네임노드 최적화를 통한 마스터 메모리 절감**

**1 팀원 소개**

컴퓨터공학부 201911265 이석진과 201910162 문무현으로 팀을 구성했습니다.

이석진은 블록 매칭 알고리즘 고안, 문무현은 파일 브로드캐스팅을 구현하였습니다.

**2 작품 소개**

람다 아키텍처에서 카프카로 제공되는 작은 데이터들을 배치 레이어에서 모아서 처리를 하기 때문에 크게 유용하지 않을 수 있습니다. 하지만 실시간 레이어에서 처리되는 실시간 데이터는 사이즈가 작습니다.  이 데이터와 통합하여 최종 사용자에게 제공하는 방식에서는 작은 파일로 처리됩니다. 람다아키텍처에서는 필연적으로 작은 사이즈의 파일들이 축적될 수 밖에 없는 구조입니다. 하둡에서는 하나의 파일은 사이즈가 작더라도 하나의 블록으로 처리됩니다. 블록의 개수가 늘어날수록 할당되는 job이 많아지며 더 많은 메모리를 사용하게 됩니다. 작은 파일을 하나의 블록으로 합쳐서 저장하므로 Job의 수를 절감할 수 있습니다. 이는 마스터노드의 메모리를 크게 절감할 수 있습니다.

**2.1 핵심 문제**

**이전의 목표** : 마스터 노드의 네임노드 메모리를 사용량을 줄이는 것

**변경된 목표** : Job에 할당되는 메모리를 절감하여 마스터 노드의 메모리를 절감시키는 것

**2.2 해결 방법**

작은 파일을 하나의 블록으로 통합 함으로서 블록의 개수를 줄입니다.

줄어든 파일의 개수만큼 마스터 노드의 메모리가 절약됩니다.

해당 블록에 대한 메타데이터는 마스터가 저장한후 슬레이브로 전달합니다.

슬레이브 노드의 메모리에 메타데이터 정보를 가지도록 하여서 슬레이브에서 메타데이터를 활용합니다.

**2.3 SW 설계**

**기능 요구사항** : 블록매칭을 통한 블록 수를 절감한 후, 해당 블록의 메타데이터를 전달합니다.

**비기능 요구사항** : 얼마만큼 많은 소형파일들을 저장하며 슬레이브 메모리를 어떻게 절약할 수 있을 것인가? 파일 1개당 4바이트를 활용하여서 위치를 지정합니다. 12비트는 블록 단편화에 사용하며 20비트는 저장된 블록을 지정하기 위해 사용합니다. 64MB 단위로 블록시 64MB/(2^12) = 16KB입니다. 총 저장가능한 용량은 64MB\*2^20 = 64TB 저장이 가능합니다. 블록사이즈에 따라서 달라지며 사용자의 기호에 맞게 단편화 사이즈도 조절이 가능합니다.

**SW 설계도** : 기본적으로 하둡의 HDFS 구성과 동일합니다. 메타데이터를 브로드캐스팅하는 과정을 콜라보레이션 다이어그램으로 표현하였으며, 서버와 클라이언트의 클래스 다이어그램을 추가하였습니다.

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**최종 산출물 구성**

**인프라 레이어** : hypervisor VM에 Ubuntu를 활용하여서 진행예정입니다.

**플랫폼 레이어** : 자바와 하둡을 활용할 예정입니다.

**어플리케이션 레이어** : 개발 모듈이름은 BC4RMM(block compression for reduction master’s memory)이 자바와 하둡만을 활용합니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**3 예상되는 문제**

****Risk Analysis****

**메타데이터 전달** : 처음에 어떻게 전송하고 어떻게 이 메타데이터를 업데이트 할 건인가?

**전달된 파일을 처리하는 방법** : 블럭 내에서 원하는 파일위치를 어떻게 찾을 것인가?

**블록 매칭 알고리즘** : 블록 사이즈와 파일 사이즈에 따라 어떻게 최적화 할 수 있는가?

****Risk Reduction Plan****

**슬레이브로 메타데이터 전달** : Java 소켓 **통신을 통해서 처리합니다.**

**전달된 파일을 처리하는 방법** : 4바이트의 정규화 된 정보를 처리하여 최적화합니다.

**블록 매칭 알고리즘** : Greedy 알고리즘을 적용합니다.

**4 코드 소개**

**4.1 서버 코드 소개**

사용자의 디렉토리로 부터 파일을 읽고 작은 파일들을 모아서 병합합니다. 병합된 파일은 .block 형식으로 산출됩니다. block 파일 내에서 데이터 위치를 찾는 Metadata.txt 파일을 생성합니다. 메타데이터 파일은 소켓을 통해 Slave 노드들에게 전달합니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

사용자의 디렉토리로부터 읽은 파일들을 블록 사이즈가 가득찰 때까지 병합합니다.

블록 사이즈가 가득차면 블록번호를 +1한 후 다음 블록에서 병합을 진행합니다.

블록에 남는 공간이 있는 경우 메타데이터를 통한 데이터 위치를 쉽게 파악하기 위해서

패딩으로 처리합니다. 한 블록의 병합이 끝나면 블록의 메타데이터를 Metadata.txt 파일에 저장합니다. 앞쪽 12비트는 블록 내에서 파일의 위치를, 뒤쪽 20비트는 데이터가 저장된 블록의 위치를 표현합니다. 사용자가 입력한 파일을 모두 처리할 때까지 반복합니다. 작업이 끝나면 메타데이터 파일을 Slave에게 소켓을 통해 전송합니다.

**4.2 클라이언트 코드 소개**

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**FileDataLoader 클래스**

* 파일을 줄 단위로 읽습니다.
* 각 줄을 공백을 기준으로 분리하여 파일 이름과 메타데이터를 얻습니다.
* 메타데이터를 사용하여 블록 이름과 파일 위치를 계산합니다. (12비트로 블록 내 위치, 20비트로 저장된 블록 지정)
* 이를 바탕으로 FileInfo 객체를 생성하고 링크드리스트로 저장합니다.

**HadoopFileReader 클래스**

* FileDataLoader로부터 파일 정보를 받아와, Hadoop 파일 시스템(HDFS)에서 파일을 읽는 역할을 합니다.
* 주어진 파일 이름에 해당하는 FileInfo 객체를 찾습니다. (findFileInfo 메서드 사용)
* 파일 시스템 구성 및 HDFS 파일 경로를 설정합니다.
* FileInfo 객체에 저장된 파일 위치로 이동하여 파일 내용을 읽습니다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Wordcount를 metadata 파일을 통해 실행하기 위해서 하둡의 InputFormat과 RecordReader 클래스를 오버라이딩했습니다.

**5 테스트 방법**

apache hadoop 공식 문서에 따르면 하둡 파일의 최대 개수의 디폴트 설정은 1048576개입니다.

따라서 1M개의 ebook 파일 입력을 받은 후 WordCount를 수행하는 것으로 결정했습니다.

실행 중 마스터와 슬레이브 노드의 메모리 점유율과 소요 시간을 비교합니다.

**1. 테스트는 1M의 파일을 직접 입력**

**2. 1M개의 파일을.block 파일로 병합 후 입력**

**3. 동일한 크기의 단일 파일을 입력**

세가지 방법으로 진행하였습니다.

**5.1 입력 파일**

프로젝트 테스트를 위해 선택한 파일은 ebook 파일입니다.

<https://www.gutenberg.org/> 프로젝트 구텐베르크에서 ebook을 텍스트 형식으로 다운로드 받은 후 프로그램에 입력값으로 넣고 수행하였습니다.

**5.2 출력 파일**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명블록 파일 출력입니다. 남는 공간이 Substitute 문자로 패딩 처리된 것을 볼 수 있습니다.텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

메타데이터 파일 출력입니다. 데이터의 이름과 위치 정보를 가집니다.

**6 수행 결과**

1차 목표 달성은 실패했습니다. 5만개의 파일의 Wordcount를 진행할 때 마스터 노드의 메모리 사용량은 약 30GB 이상이며, 소요시간은 약 55분이었습니다. 100만개의 파일을 테스트하는 것은 600GB의 메모리가 필요하여 진행할 수가 없었습니다. 일반적인 사용자 PC의 메모리를 고려할 때 100만개의 파일의 Wordcount를 테스트하는 것은 무리가 있었습니다. 그럼에도 메모리를 최대한 확보하여 슬레이브 16GB씩 둘과 마스터 50GB로 82GB 세팅한 후 진행하였습니다. 컴퓨터 자원의 부족으로 테스트 파일의 개수를 5만개로 줄인 후 수행하였습니다.

하둡을 실행시킨 직후와 HDFS에 파일 5만개를 올린 후 메모리 사용량을 비교한 결과 메모리 사용량에 거의 차이가 없었습니다. 파일을 HDFS에 올리는 것은 메모리 사용에 거의 영향을 주지 않는 것을 알 수 있습니다.

문제는 워드카운터를 실행시 발생하였습니다. 5만개 파일의 Wordcount를 수행 시 메모리를 32기가 할당한 환경에서는 작동하지만 30기가 환경에서는 작동하지 않았습니다.

마스터 노드의 메모리 부족으로 인한 OutOfMemoryError: Java heap space 에러가 발생하였습니다.

swap 공간을 늘려보았지만 문제가 해결되지 않았습니다. HeapMemory는 Swap 공간에 할당할 수가 없어 실제 메모리 공간만으로 감당하여서 특히 컴퓨팅 자원이 많이 요구 합니다.

Wordcount를 실행 중일 때 정확한 사용량을 모니터링 프로그램에서 정확히 추정이 되지는 않습니다. 대략 3배정도 사용함을 알수 있습니다. 32기가 중 1.7기가 정도를 OS에서 사용하며 나머지 30.3기가 정도를 하둡에서 사용하므로 5만개의 JOB 즉 파일이 각각 약 630KB의 메모리를 사용하는 것을 알 수 있습니다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

이를 해결하기 위해서 Java Heap Memory Error는 하둡 설정 파일에서 mapred-site 파일을 수정하여 메모리 제한을 늘려서 수행을 하였습니다.

가상머신이 계속 죽는 문제가 발생하여서 하둡이 Safe Mode로 들어가는 문제가 있었습니다.

Namenode Safe Mode로 인해 발생하는 문제는 마스터 노드에서 safemode를 해제하여 해결하였습니다.

수행 결과 5만개의 파일을 직접 Wordcount를 수행한 경우는 메모리 사용량은 30GB 이상, 소요시간은 약 54분이었습니다.

파일을 BC4RMM을 통해 .block 형식으로 변환한 후 Wordcount를 수행한 결과 180개의 블록과 1개의 메타데이터 파일이 생성되었으며, 메모리 사용량은 200MB 미만, 소요시간은 약 23분이었습니다.

마지막으로 5만개의 파일과 동일한 크기의 단일 파일의 Wordcount를 수행한 결과는 메모리 사용량은 200MB 미만, 소요 시간은 29분이었습니다.

텍스트, 스크린샷, 폰트, 원이(가) 표시된 사진

자동 생성된 설명 5만개의 파일 결과

텍스트, 스크린샷, 폰트, 그래픽이(가) 표시된 사진

자동 생성된 설명 BC4RMM 사용 결과

텍스트, 스크린샷, 폰트, 그래픽이(가) 표시된 사진

자동 생성된 설명 단일 파일 결과

Ebook 데이터를 사용하였습니다. 5만개의 파일이 180개로 크게 줄어들었고 이 덕분에 극적인 차이가 발생하였습니다. 단일 파일로 수행하였을 때 오히려 시간이 늘어난 것은 예상 밖이었는데, 파일의 크기 때문에 하둡에서 다시 블록으로 나누는 과정이 추가되기 때문에 시간이 늘어난 것으로 보입니다. Ebook보다 큰 데이터를 사용하는 경우 테스트 결과만큼 큰 차이가 발생하지는 않겠으나, 작은 파일들이 입력으로 주어지는 데이터를 사용하는 경우 충분히 사용 가치가 있다고 생각됩니다. 파일의 개수가 많아질 수록 차지하는 메모리 용량이 사용자와 컴퓨터에게 부담이 되기 때문입니다.

하둡에서는 하나의 파일은 하나의 블록을 차지하는데, 이러한 낭비를 줄여서 부담을 줄이고 효율적으로 사용할 수 있습니다. 일반 사용자의 PC 램 크기를 고려했을 때 파일의 개수가 많은 경우 마스터 노드의 메모리로 인해 실행이 불가능할 것입니다. 이런 경우에 BC4RMM을 사용하면 실행이 불가능한 상황에서 하둡을 사용할 수 있게 됩니다. 뿐만 아니라 데이터 입력으로 큰 파일과 작은 파일이 섞여서 들어올 때도 효율적일 수 있습니다. 입력 파일들이 모두 블록사이즈보다 큰 경우에는 차이가 없지만, 네임노드의 블록사이즈보다 작은 파일이 많을 수록 효율적입니다.

**7 참고자료**

참고자료로 클라우데라 하둡 세팅 가이드라인과 아파치 하둡 공식문서를 활용하였습니다.

클라우데라 하둡 세팅 가이드라인 (2021년 자료)

<https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.5.0/bk_command-line-installation/content/configuring-namenode-heap-size.html>

Apache Hadoop 공식 문서

<https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml>