REPORT

빅데이터검색 3조 기말레포트

과목명	빅데이터검색
담당교수	김영훈 교수님
학생이름	박준우, 오예성
학과	인공지능학과, 해양융합공학과
학번	2021006253, 2018047938
제출일	2023.06.07

HANYANG UNIVERSITY

1. Moudle1 - Tokenizer

① 스텝별 설명

```java
public class HanyangSETokenizer implements Tokenizer{
    private final Analyzer analyzer=new SimpleAnalyzer();
    private final PorterStemmer stemmer=new PorterStemmer();

    public HanyangSETokenizer(){
    }

    /**
     * Tokenizes the input text and returns the list of tokens.
     */
    @Override
    public List<String> split(String text){
        List<String> tokens=new ArrayList<>();
        try (TokenStream stream=analyzer.tokenStream(null, new StringReader(text))){
            stream.reset();
            CharTermAttribute ta=stream.addAttribute(CharTermAttribute.class);
            while (stream.incrementToken()){
                stemmer.setCurrent(ta.toString());
                stemmer.stem();
                tokens.add(stemmer.getCurrent());
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return tokens;
    }

    /**
     * Closes the analyzer instance.
     */
    @Override
    public void clean(){
        analyzer.close();
    }

    /**
     * Initializes the tokenizer instance.
     */
    @Override
    public void setup(){
        // no-op
    }
}
```

1) split 메소드

analyzer를 사용하여 텍스트를 받아서 토큰으로 분리하고, 이 과정에서 불필요한 공백이나 특수 문자를 제거하여 단어의 기본 형태인 어간을 추출하는 과정을 거칩니다. 이후 stemmer를 사용하여 각 토큰의 원형을 찾은 후 결과를 반환합니다.

2) Clean 메소드

analyzer 인스턴스를 닫으 사용한 자원을 해제합니다.

3) Setup 메소드

Setup 메소드는 토크나이저를 초기화합니다.

② 어려웠던 점과 새롭게 알게 된 것

Tokenizer의 경우 뼈대코드 및 예제코드가 자세하게 나와있어서 어려웠던 점은 크게 없었습니다. 이 과제를 수행하면서 porterstemmer가 접두사나 접미사를 제거한 영어 어간 추출 알고리즘임을 알게 되었고, analyzer는 텍스트 전처리와 관련된 자연어 처리 도구로 토큰화와 정규화 불용어 처리등의 기능을 한다는 것을 알게 되었습니다.

③느낀점

막상 이론만 들었을 때는 어떻게 구현해야 할 지 감도 안 잡히고 어려워 보이는데 lucene과 porterstemmer와 같은 라이브러리 덕분에 짧은 코드로도 해당 기능을 수행할 수 있다는 점이 놀라웠습니다.

2. Module2 - ExternalMergeSort

①스텝별 설명

1) sort 메소드

```java
public class HanyangSEExternalSort implements ExternalSort {
    private int nblocks;//내부구현 및 다른 접근 방지로 private
    private int BLOCKSIZE = 1024*8;
    private int N_BLOCKS = 500;
    private int TOTALSIZE = 4000000;

    @Override

    public void sort(String infile,String outfile,String tmpdir,int blocksize,int nblocks)throws IOException{
        this.nblocks=nblocks;
        // 초기 실행을 위한 임시파일 생성
        File temp=new File(tmpdir+File.separator +"0");
        if (!temp.exists()){
            temp.mkdir();
        }
        //1)initial phase
            try(DataInputStream dis=new DataInputStream(new BufferedInputStream(new FileInputStream(infile), (blocksize * nblocks)/12))){
                int runSize=blocksize;//실행할 블록의 크기
                int runCount=0;//실행 횟수
                ArrayList<MutableTriple<Integer,Integer,Integer>>dataArr=new ArrayList<>();
                String runFileBase = tmpdir + File.separator + "0" + File.separator;
                BufferedOutputStream bos = new BufferedOutputStream(null);
                while (dis.available()>0){
                    for (int i=0;i<runSize&& dis.available()>0;i++) {
                        int left=dis.readInt();//왼쪽 단어번호 읽기
                        int middle=dis.readInt();//중간 문서번호 읽기
                        int right=dis.readInt();//오른쪽 워치번호 읽기
//                          System.out.println("Triple: (" + left + ", " + middle + ", " + right + ")");//확인용 코드
                        dataArr.add(new MutableTriple<>(left, middle, right));
                    }

                    dataArr.sort(Comparator.comparingInt(MutableTriple<Integer,Integer,Integer>::getLeft).
                            thenComparingInt(MutableTriple<Integer, Integer, Integer>::getMiddle)
                            .thenComparingInt(MutableTriple<Integer, Integer, Integer>::getRight));

                    String runFile = runFileBase + "runcount" + runCount + ".data";
                    bos = new BufferedOutputStream(new FileOutputStream(runFile), (blocksize * nblocks)/12);
                    try (DataOutputStream dos = new DataOutputStream(bos)) {
                        for (MutableTriple<Integer,Integer,Integer> triple : dataArr) {
                                dos.writeInt(triple.getLeft());
                                dos.writeInt(triple.getMiddle());
                                dos.writeInt(triple.getRight());
                        }
                    }
                    runCount++;//실행횟수 1씩 추가
                }
                bos.close();//bos 메모리 종료
        //2)n-way merge
        _externalMergeSort(tmpdir,outfile,0);//외부합병정렬메소드 호출, 임시디렉토리와 출력파일,초기번호 0 전달
            }
    }
```

sort 메소드는 외부 정렬을 시작하는 곳으로 입력 파일을 블록 단위로 나누어 임시 파일에 저장하는 단계입니다. 입력 파일을 읽고, 정렬할 각 블록에 대해 초기 실행 단계를 수행합니다. 각 실행은 입력 파일의 일부 블록을 읽고, 각 블록을 정렬한 다음 임시 파일에 저장하는 과정입니다. 임시파일은 runCount 즉 실행 횟수를 값으로 하는 변수로써 파일명을 지정하여 구분하였습니다.

2) _externalMergeSort 메소드

```
private void _externalMergeSort(String tmpDir,String outputFile,int step)throws IOException{
    File[] fileArr=(new File(tmpDir+File.separator+String.valueOf(step))).listFiles();
    if (fileArr==null||fileArr.length==0) {
        return;
    }

    if (fileArr.length<=this.nblocks-1){
        List<DataInputStream> disList=new ArrayList<>();
        for (File f:fileArr){
            disList.add(new DataInputStream(new BufferedInputStream(new FileInputStream(f))));
        }
        n_way_merge(disList,outputFile);

        for (DataInputStream dis :disList){
            dis.close();
        }
    }
    else {
        int cnt=0;
        int fileNum = 0;
        List<DataInputStream> disList=new ArrayList<>();

        for (File f:fileArr) {
            disList.add(new DataInputStream(new BufferedInputStream(new FileInputStream(f))));
            cnt++;
            File nextStepDir=new File(tmpDir+File.separator+(step+1));
            if (!nextStepDir.exists()){
                nextStepDir.mkdir();
            }
            if (cnt==this.nblocks- 1){
                String nextOutputFile;
                nextOutputFile = tmpDir + File.separator + (step + 1) + File.separator + "runcount" + fileNum + ".data";
                n_way_merge(disList,nextOutputFile);
                for (DataInputStream dis:disList){
                    try {
                        dis.close();
                    } catch (IOException e){
                        e.printStackTrace();
                    }
                }
                disList.clear();//모든 데이터 리스트 초기화
                cnt = 0;//실행횟수 0으로 초기화
                fileNum++;
            }
        }
            String nextOutputFile;
            nextOutputFile = tmpDir + File.separator + (step + 1) + File.separator + "runcount" + fileNum + ".data";
            n_way_merge(disList,nextOutputFile);
            for (DataInputStream dis:disList){
                try {
                    dis.close();
                } catch (IOException e){
                    e.printStackTrace();
                }
            }
        _externalMergeSort(tmpDir,outputFile,step+1);
    }
}
```

_externalMergeSort 메소드는 tmpDir 디렉토리에서 파일 목록을 가져옵니다. 이 블록들을 한 번에 nblocks개씩 묶어서 병합합니다. 이렇게 하면 입력으로 주어진 모든 블록들이 더 크고 정렬된 블록들로 병합되게 됩니다. 이 과정은 재귀적으로 수행되며 모든 블록이 하나로 병합될 때까지 계속됩니다.

3) n_way_merge 메소드

```
public void n_way_merge(List<DataInputStream> files,String outputFile)throws IOException{

    PriorityQueue<DataManager> queue = new PriorityQueue<>(files.size(),
            Comparator.comparing(DataManager::getTuple));


    for (DataInputStream dis:files){
        DataManager dm=new DataManager(dis);
        if (!dm.isEOF){
            queue.add(dm);
        }
    }
    try (DataOutputStream dos=new DataOutputStream(new BufferedOutputStream(new FileOutputStream(outputFile)))){
        while(queue.size()!=0){
            DataManager dm=queue.poll();
            MutableTriple<Integer,Integer,Integer>tmp=dm.tuple;
            dos.writeInt(tmp.getLeft());//트리플 왼쪽 단어번호를 dos에 쓰기
            dos.writeInt(tmp.getMiddle());//트리플 중간 문서번호를 dos에 쓰기
            dos.writeInt(tmp.getRight());//트리플 오른쪽 위치번호를 dos에 쓰기
            dm.readNext();
            if (!dm.isEOF){//파일 비어있지않으면
                queue.add(dm);//큐에 datamager추가
            }
        }
    }
}
```

n_way_merge 메소드는 여러 정렬된 파일을 한 번에 병합합니다.

우선 순위 큐를 사용하여 병합되어야 하는 블록에서 가장 작은 값 또는 튜플을 찾고 비교하여 가장 작은 요소를 출력 파일에 쓰는 방식으로 작동합니다.

이 과정은 모든 파일의 모든 요소가 사용될 때까지 계속됩니다. 이를 통해 출력 파일에는 정렬된 데이터만 존재하게 됩니다.

4) DataManager 클래스

```
public static class DataManager{
    public boolean isEOF=false;//파일의 끝 false로 초기화
    private DataInputStream dis=null;//dis null로 초기화
    public MutableTriple<Integer,Integer,Integer> tuple=new MutableTriple<Integer,Integer,Integer>(0,0,0);
    public DataManager(DataInputStream dis2)throws IOException{
            this.dis=dis2;
            isEOF=!readNext();//dis의 끝에 남은 데이터 없는지 확인

    }

    private boolean readNext()throws IOException{//다음 튜플 읽기

        try {
            tuple.setLeft(dis.readInt());//트리플 왼쪽 단어번호를 dos에 읽기
            tuple.setMiddle(dis.readInt());//트리플 왼쪽 단어번호를 dos에 읽기
            tuple.setRight(dis.readInt());//트리플 왼쪽 단어번호를 dos에 읽기
            return true;
        }catch(EOFException e) {//파일의 끝인 경우
            isEOF = true;//isEOF가 true가 됨
            return false;//false반환
        }
    }
    public MutableTriple<Integer, Integer, Integer> getTuple() {
        return tuple;
    }
    public boolean isEOF() {
        return isEOF;
    }
}
```

DataManager 클래스는 파일의 데이터 스트림으로부터 튜플을 읽고 관리하는 데 사용되며 데이터 매니저는 다음에 읽어야 할 run의 인덱스를 추적하기 위해서 구현하였습니다.

이 클래스의 객체는 파일에서 다음 튜플을 읽어올 때마다 업데이트되며, 파일이 끝날 경우에는 EOF 상태를 설정합니다.

② 어려웠던 점과 새롭게 알게 된 것

input데이터와 output데이터의 크기가 같아야 하는데 자꾸만 output데이터의 크기가 input에 비해 작게 나왔었습니다. 알고보니 output데이터를 만들 때 새로운 파일을 생성하는 것이 아니라 기존 파일에 덮어쓰는 형식으로 쓰다보니 이런 문제가 발생하였습니다.

그리고 데이터를 읽어들이는 속도가 생각보다 오래 걸렸고 작년 기준 평균 실행속도는 달성했지만 최고 수준의 속도에는 아직 도달하지 못하였기에 최적화를 위해 노력하였습니다. 객체 생성을 최소한으로 하고 객체 및 버퍼를 재사용하는 방향으로 하여 최적화를 위해 노력했지만, 1~2초 정도의 미미한 개선만 있을 뿐 눈에 띄게 실행시간이 줄지는 않았습니다. 그리고 최적화를 위해 outputstream 데이터를 flush하는 과정을 넣었었는데 조교님께서 말씀해주시기를 flush를 하게 되면 스트림을 할 때마다 디스크에 들어갔다 나왔다를 반복해야 되기 때문에 오히려 시간이 오래 걸릴 수도 있다고 말씀해 주셨고 이를 처음 알게 되었습니다.

③느낀점

앞의 과정을 거친 이후에는 어디를 더 최적화해야 시간이 줄어들 수 있는지 가늠이 되질 않았고 실행시간이 더 이상 안 줄어드는 것을 보고 이건 코드의 논리적 구조 때문이 아닐까 하는 생각을 했습니다.

그리고 작년 기준의 성적들은 알고 있지만 올해 다른 조들의 실행속도 결과의 경우 알 수가 없다보니 저희 팀의 수준이 어느 정도인지 객관화를 할 수 없었다는 점이 더 초조했던 것 같습니다.

3. Module3 – BPlus Tree

1. 과정

A. Open

open메소드는 메타파일, 트리파일, 블록사이즈, 블록 개수를 변수로 받아 실행됩니다. 메타 파일에는 루트 노드의 위치, fanout, 블록 사이즈가 저장되어 있습니다.

파일에 저장된 정보를 읽기 위해 randomaccessfile을 만들어 사용합니다. 이때 tree를 처만드는 경우에는 root 위치, fanout, 블록 사이즈를 초기값으로 설정합니다. tree 정보가 있다면 mata 파일에서 mata 정보를 읽어 사용합니다.

트리를 최초로 만드는 경우에는 root node를 새로 생성하고, 아닐 경우에는 root 위치를 기반으로 root Node를 파일에서 찾아 생성합니다.

```
public void open(String metafile, String treefile, int blocksize, int nblocks) throws IOException {
    this.treepath = treefile; // 트리 파일
    this.metapath = metafile; // 메타 파일
    this.blockSize = blocksize; // 노드 크기
    this.nodecnt = nblocks; // 블록 개수

    // 파일 입출력
    this.meta = new RandomAccessFile(metafile,  mode: "rw");
    this.tree = new RandomAccessFile(treefile,  mode: "rw");

    // 트리가 존재할 경우
    if ( meta.length() > 4L) {
        this.mode = 0;
        if(meta.getFilePointer() <= meta.length() - 4L) {  //정수형 읽기에 여유 공간 있는지 확인
            this.rootPos = this.meta.readInt();
        }
        this.fanout = this.meta.readInt();
        this.blockSize = this.meta.readInt();
    // 트리를 처음 만들 경우
    } else {
        this.mode = 1;
        this.rootPos = 0;
        this.fanout = blockSize / 8  - 1;//(Integer.SIZE / 4) 하나의 노드에 들어갈 수 있는 int값의 개수 -1은
    }

    cache = new LRU(nblocks, this.mode);// 캐시 초기화 캐시 크기를 nblock로, 모드 설정

    if (tree.length() > 0)//트리 길이 0보다 큼
        root = MakeNode(rootPos);//루트노드를 만듦
    else root = new Node( isLeaf: true,  pos: -1);//새로운 노드 생성 후 루트로 설정

    bytes = new byte[this.blockSize];//크기 바이트 배열 초기화
    buffer = ByteBuffer.wrap(bytes);
}
```

B. Cache 생성

이후 LRU 기법을 사용한 cache를 생성합니다. 이를 통해 삽입, search 과정에서 파일에 직접 접근을 최소화하며, insert 과정에서는 write를 최소화할 수 있습니다.

```
public class LRU {//주어진 용량 초과시 최근에 사용하지 않은 항목 제거

    2 usages
    int mode;//캐시 모드
    2 usages
    int capacity;//캐시 용량
    8 usages
    LinkedHashMap<Integer, Data> map;//캐시에 저장된 데이터 보관하는 장소
    1 usage
    public LRU(int capacity, int mode) {
        this.capacity = capacity;
        this.mode = mode;//맵의 capacity는 맵의 초기 크기,75f는 로드팩터 0과 1사
        this.map = new LinkedHashMap<Integer, Data>(capacity,  loadFactor: .7
            @Override
            protected boolean removeEldestEntry(Map.Entry<Integer, Data>
                if (size() == capacity)//캐시용량이 가득차면
                    try {
                        LRU.this.RemoveNode(eldest);
                    } catch (IOException e) {

                    }
                return size() > LRU.this.capacity;//캐시용량 초과여부 반환
            }
        };
    }
```

C. Insert

key와 value를 root 노드에 삽입하면 node 클래스 안에 정의된 함수에 의해 값이 삽입됩니다.

leaf 노드에서는 key 값을 기준으로 삽입할 위치를 찾아 넣습니다.

non leaf 노드에서는 해당 키를 삽입할 자식노드를 찾고 key, value를 넣게 됩니다. 이 때 자식 노드가 가득 차게 되면 split 함수를 통해 분할해줍니다.

삽입 이후에는 삽입한 노드를 cache에 저장합니다.

```java
@Override
public void insert(int key, int value) throws IOException {
    root.InsertValue(key, value);
}
```

```java
public void InsertValue(int key, int value) throws IOException {
    if (this.isLeaf) {

        int pos = Arrays.binarySearch(keys, fromIndex: 0, nKeys, key);//키가 삽입될 위치 탐색
        int valueIndex = pos >= 0 ? pos : -pos - 1;//동일 키가 존재하면 해당 위치에, 아니면 삽입할

        for (int idx = this.nKeys - 1; idx >= valueIndex; idx--) {//위치를 찾은 이후에 모든 키와
            this.keys[idx + 1] = this.keys[idx];
            this.values[idx + 1] = this.values[idx];
        }

        this.nKeys++;//키의 개수 증가
        this.keys[valueIndex] = key;//새 키 삽입
        this.values[valueIndex] = value;//새 값 삽입
        cache.SetNode(this.pos, this);//캐시에 해당 노드 설정

    } else {
        Node childNode = GetChild(key, isInsert: 1);//해당 키가 삽입될 자식 노드 탐색
        childNode.InsertValue(key, value);//그 노드에 키와 값 삽입

        if (childNode.nodeIsFull()) {//자식노드 풀이면 분할
            int leftdata = childNode.Split();
            int pos = nodecnt * blockSize;
            InsertNode(leftdata, pos);//분할 노드의 첫 키를 해당 노드에 삽입
            cache.SetNode(this.pos, this);
        }
    }

    if (root.nodeIsFull()) {
        nodecnt++;//루트 노드가 가득참
        Node newRootNode = new Node( isLeaf: false, pos: -1);//리프가 아니고 새로운 노드
        newRootNode.keys[0] = Split();//루트 노드 분할
```

D. Split

split 시 [fanout/2]를 기준으로 new node에 key 값을 할당하고 nkey를 증가시켜 줍니다. 이때 현재 노드의 leaf node 여부에 따라 값을 다르게 할당합니다.

```java
public int Split() throws IOException {
    nodecnt++;//분할하므로 노드개수 1 증가
    Node newNode = new Node(this.isLeaf, pos: -1);//새로운 노드 생성 같은 타입의 노드 생성,위치는 새

    int mid = nKeys / 2;//분할할 키
    int end = nKeys;

    int leafInt = this.isLeaf ? 0 : 1;//리프노드 여부 판단 리프면 0 아니면 1
    for (int idx = 0; idx < end - mid - leafInt; ++idx)
        newNode.keys[idx] = keys[idx + mid + leafInt];//키를 새 노드로 이동시킴

    newNode.nKeys = end - (mid + leafInt);//새 노드의 개수 조정
    this.nKeys -= end - (mid + leafInt);//해당 노드의 키 개수 조정

    for (int idx = 0; idx < end - mid; ++idx)//값을 새 노드로 이동
        newNode.values[idx] = values[idx + mid + leafInt];

    cache.SetNode(newNode.pos, newNode);//캐시에 새 노드를 설정
    return keys[mid];//분할된 첫 번째 키를 반환 -> 상위 노드에 삽입되며, 분할되는 노드들을 분리하는 역할
```

E. 새로운 root 생성

이후 root 노드가 가득 찼다면, 새로운 노드를 생성해 root node로 지정해 줍니다.

```
if (root.nodeIsFull()) {
    nodecnt++;// 노드를 한개 늘려줌
    Node newRootNode = new Node( isLeaf: false, pos: -1);//리프가 아닌 새로운 노드
    newRootNode.keys[0] = Split();//루트 노드 분할
    newRootNode.nKeys++;//키의 개수 증가
    newRootNode.values[0] = this.pos;//새로운 루트노드 첫번 째 자식
    newRootNode.values[1] = nodecnt * blockSize;//새로운 루트노드의 두 번째 자식
    root = newRootNode;
}
```

F. Cache를 이용한 데이터 저장

만약 insert 과정에서 cache 크기보다 많은 값이 삽입되면 사용이 가장 오래된 노드의 정보를 파일에 적게 됩니다. 이때 데이터를 파일에 빠르게 쓰기 위해 ByteBuffer를 사용해 데이터를 file에 적습니다.

```
public void writeData(int offset) throws IOException {

    buffer.clear();
    buffer.putInt(this.isLeaf ? 1 : 0);//리프노드 여부 판단 리프면 1 아니면 0
    buffer.putInt(this.nKeys);//키의 개수를 버퍼에 작성

    for (int idx = 0; idx < this.nKeys; ++idx) {//각 키와 대응하는 값들을 버퍼에 작성
        buffer.putInt(this.keys[idx]);
        buffer.putInt(this.values[idx]);
    }

    if (!isLeaf && nKeys > 0)//리프노드 아니면서 키 존재시 추가값을 버퍼에 작성
        buffer.putInt(this.values[nKeys]);//마지막 키보다 큰 키들을 가진 자식 노드를 가리킴

    tree.seek(this.pos);//트리파일에서 해당 노드의 위치로 이동
    tree.write(bytes);
}
```

G. Search

탐색 과정에서는 root 노드부터 Getchild 함수를 이용해 leaf 노드까지 재귀적으로 탐색합니다. leaf 노드를 찾으면 key값에 해당하는 값을 찾습니다.

```
public int Search(int key) throws IOException {

    if (this.isLeaf) {
        int pos = Arrays.binarySearch(keys, fromIndex: 0, nKeys, key);//키 배열에서 이진탐색으로 키를 찾음
        return pos >= 0 ? values[pos] : -1;//키가 있다면 해당 포지션을 반환하고 아니면 음수 반환
    } else {
        Node node = GetChild(key, isInsert: 0);//노드가 내부 노드인 경우, 해당 키가 위치해야 할 자식 노드를 가져옴
        return node.Search(key);//가져온 자식노드에서 재귀적으로 탐색수행
    }
}
```

search 시에도 cache를 이용하게 되는데 cache에 찾고 있는 값이 없는 경우에는 해당 노드를 만들어 cache에 삽입하고 노드가 cache에 존재하면 찾아 반환합니다.

```
public Node GetNode(int key) throws IOException {
    if (map.containsKey(key))//키가 캐시 맵에 존재하면
        return map.get(key).node;//해당 노드를 반환 즉 빠른 반환
    else {
        Node node = MakeNode(key);//새 노드를 만들고 캐시에 추가한 후에 반환
        SetNode(key, node);
        return node;
    }
}
```

H. Close

삽입, 삭제 과정이 끝나고 나면 루트 노드의 데이터를 파일에 작성하고 cache에 존재하는 모든 노드들의 정보를 tree 파일에 하나씩 저장합니다.

```
public void close() throws IOException {
    root.writeData(root.pos);//루트 노드의 데이터를 루트 노드 위치에 작성
    cache.Flush();//캐시의 데이터를 디스크에 작성
    tree.close();//파일 닫기

    meta.writeInt(root.pos);//메타 파일에 루트 노드의 위치 작성
    meta.writeInt(fanout);//메타파일에 fanout 작성
    meta.writeInt(blockSize);//메타파일에 노드 크기 작성
    meta.close();//메타파일 닫기
}
```

2. 어려웠던 점

A. 파일을 읽고 쓰는 RandomAccessfile과 ByteBuffer에 대한 개념을 이해

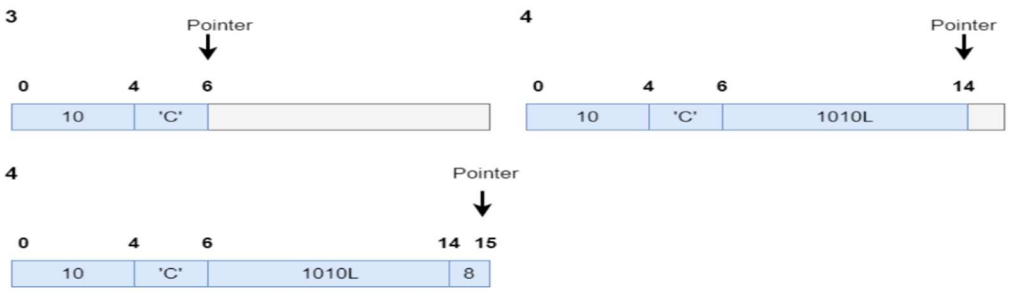
external merge sort 모듈에서 io다루긴 했지만, 다른 특성을 가진 randomAccessfile과 byteBuffer의 개념을 처음 접해 개념을 이해하는 데 시간이 걸렸습니다.

이를 이해하기 위해 오라클에서 제공하는 java document를 참고했습니다.

RandomAccessfile

1. 데이터 쓰기 - write()

파일의 현재 포인터에 내용을 적습니다. 이를 이미지로 표현하면 다음과 같습니다.



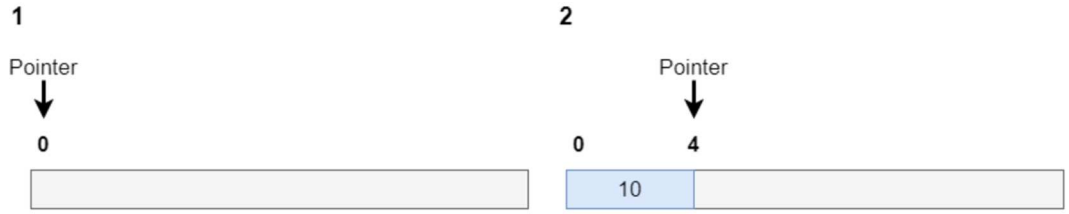
2. 데이터 읽기 - read()

파일의 현재 포인터부터 내용을 읽습니다. 다른 형식을 read뒤에 붙여 형식에 따라 읽어올 수 있습니다.

readInt(), readChar(), readLong(), readByte()

파일을 읽거나 쓰고 난 뒤에는 파일 포인터의 위치가 크기에 맞게 변환됩니다.

ex) int 형 10을 파일에 적고 난 뒤 -> 포인터는 4를 가르킴



3. 파일 포인터 변경 - seek()

파일 포인터를 pos 위치로 변경한다.

원하는 위치의 데이터를 읽어오려면 파일 포인터를 그 위치로 변경하고 read 하거나 wirte 합니다.

ByteByuffer

바이트 데이터를 저장하고 읽는 저장소, 배열을 멤버변수로 가지며, 배열에 대한 읽기/쓰기를 지원합니다.

capacity

- 버퍼에 저장할 수 있는 데이터의 최대 크기

position

- 읽기/쓰기의 작업 위치를 나타냄

- 데이터를 읽거나 쓸 경우 자동으로 크기만큼 증가

limit

- 읽고 쓸 수 있는 버퍼 공간의 최대치

생성

```
ByteBuffer byteBuffer = ByteBuffer.warp(array)
```

메소드

1. 데이터 쓰기 - put()

byteBuffer에 데이터를 할당합니다.

2. 데이터 읽기 - get()

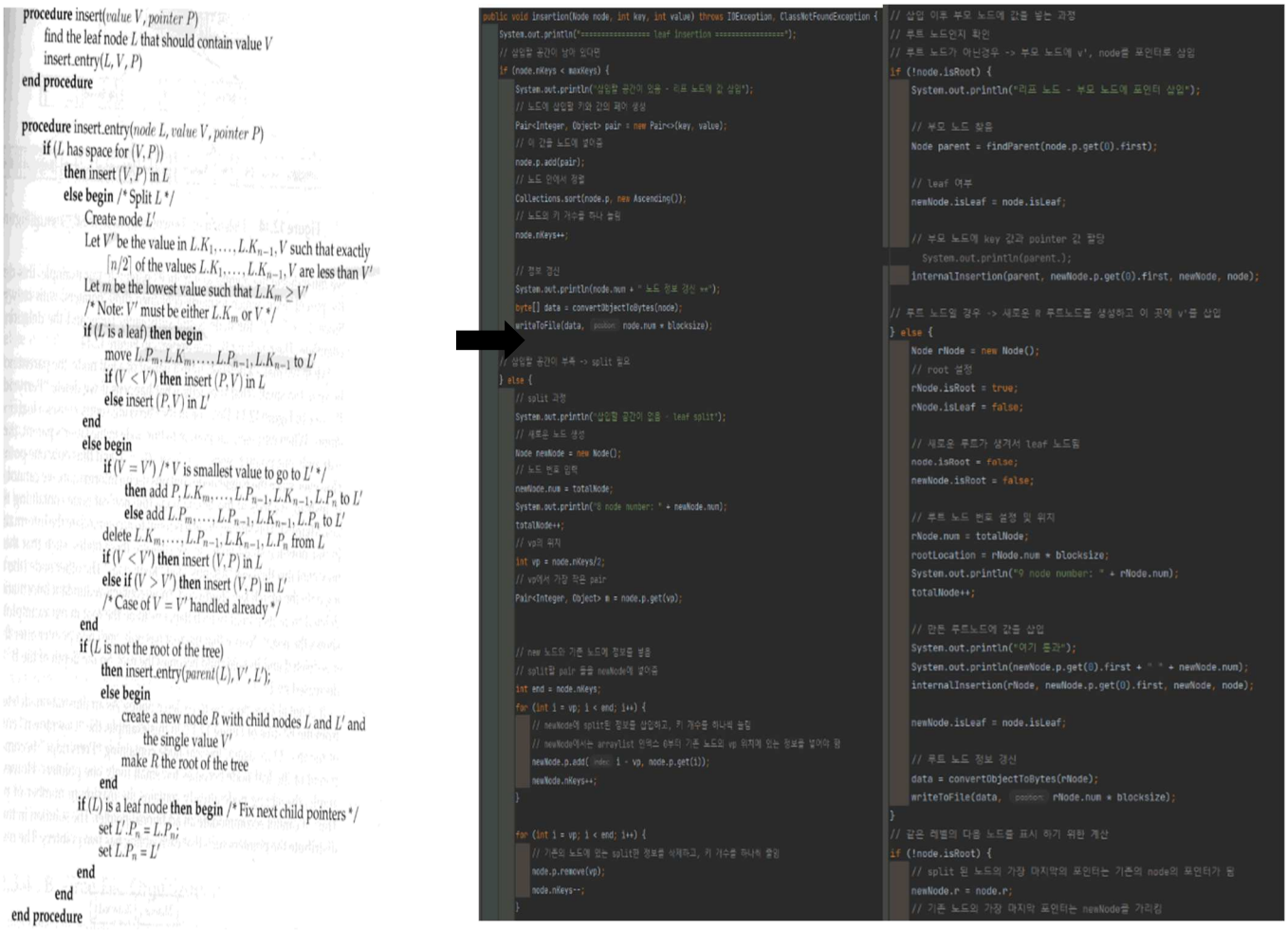
warp 되어 있는 배열에 저장된 데이터를 불러옵니다.

3. 버퍼 초기화 - clear

buffer를 클리어 합니다. 위치를 0으로 설정하고, limit과 capacity를 동일하게 설정합니다.

B. leaf node와 non leaf 노드의 insert, split 과정이 조금씩 달라 코드를 효율적으로 사용하면서 구현하기 어려웠습니다.

처음에는 주어진 pseudo코드를 따라 코드를 구성했습니다. 코드가 길어지며 가독성이 조금 떨어지고, 로직 흐름을 한 번에 이해하기 어려웠습니다.
특히, split이 일어날 경우에 leaf node와 non leaf node에서 조금 다른 차이점이 있는데 이를 구분하는 코드를 작성하며 로직을 더 직관적으로 이해하기 어려워졌습니다.



```
procedure insert(value V, pointer P)
    find the leaf node L that should contain value V
    insert_entry(L, V, P)
end procedure

procedure insert_entry(node L, value V, pointer P)
    if (L has space for (V, P))
        then insert (V, P) in L
        else begin /* Split L */
            Create node L'
            Let V' be the value in L.K_1, ..., L.K_{n-1}, V such that exactly
                ⌈n/2⌉ of the values L.K_1, ..., L.K_{n-1}, V are less than V'
            Let m be the lowest value such that L.K_m ≥ V'
            /* Note: V' must be either L.K_m or V */
            if (L is a leaf) then begin
                move L.P_m, L.K_m, ..., L.P_{n-1}, L.K_{n-1} to L'
                if (V < V') then insert (P, V) in L
                else insert (P, V) in L'
            end
            else begin
                if (V = V') /* V is smallest value to go to L' */
                    then add P, L.K_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n to L'
                    else add L.P_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n to L'
                delete L.K_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n from L
                if (V < V') then insert (V, P) in L
                else if (V > V') then insert (V, P) in L'
                /* Case of V = V' handled already */
            end
            if (L is not the root of the tree)
                then insert_entry(parent(L), V', L');
                else begin
                    create a new node R with child nodes L and L' and
                        the single value V'
                    make R the root of the tree
                end
            if (L) is a leaf node then begin /* Fix next child pointers */
                set L'.P_n = L.P_n;
                set L.P_n = L'
            end
        end
end procedure
```

```
public void insertion(Node node, int key, int value) throws IOException, ClassNotFoundException {
    System.out.println("================ leaf insertion ================");
    // 삽입할 공간이 남아 있다면
    if (node.nKeys < maxKeys) {
        System.out.println("삽입할 공간이 있음 - 리프 노드에 값 삽입");
        // 노드에 삽입할 키와 값의 페어 생성
        Pair<Integer, Object> pair = new Pair<>(key, value);
        // 이 값을 노드에 넣어줌
        node.p.add(pair);
        // 노드 안에서 정렬
        Collections.sort(node.p, new Ascending());
        // 노드의 키 개수를 하나 늘림
        node.nKeys++;

        // 정보 갱신
        System.out.println(node.num + " 노드 정보 갱신 ==");
        byte[] data = convertObjectToBytes(node);
        writeToFile(data,  position: node.num * blocksize);
    // 삽입할 공간이 부족 -> split 필요
    } else {
        // split 과정
        System.out.println("삽입할 공간이 없음 - leaf split");
        // 새로운 노드 생성
        Node newNode = new Node();
        // 노드 번호 입력
        newNode.num = totalNode;
        System.out.println("새 node number: " + newNode.num);
        totalNode++;
        // vp의 위치
        int vp = node.nKeys/2;
        // vp에서 가장 작은 pair
        Pair<Integer, Object> m = node.p.get(vp);


        // new 노드와 기존 노드에 정보를 넣음
        // split할 pair 들을 newNode에 넣어줌
        int end = node.nKeys;
        for (int i = vp; i < end; i++) {
            // newNode에 split된 정보를 삽입하고, 키 개수를 하나씩 늘림
            // newNode에서는 arraylist 안덱스 0부터 기존 노드의 vp 위치에 있는 정보를 넣어야 함
            newNode.p.add( index: i - vp, node.p.get(i));
            newNode.nKeys++;
        }

        for (int i = vp; i < end; i++) {
            // 기존의 노드에 있는 split한 정보를 삭제하고, 키 개수를 하나씩 줄임
            node.p.remove(vp);
            node.nKeys--;
        }
```

```
        // 삽입 이후 부모 노드에 값을 넣는 과정
        // 루트 노드인지 확인
        // 루트 노드가 아닌경우 -> 부모 노드에 v', node를 포인터로 삽입
        if (!node.isRoot) {
            System.out.println("리프 노드 - 부모 노드에 포인터 삽입");

            // 부모 노드 찾음
            Node parent = findParent(node.p.get(0).first);

            // leaf 여부
            newNode.isLeaf = node.isLeaf;

            // 부모 노드에 key 값과 pointer 값 할당
            System.out.println(parent.);
            internalInsertion(parent, newNode.p.get(0).first, newNode, node);

        // 루트 노드일 경우 -> 새로운 R 루트노드를 생성하고 이 곳에 v'를 삽입
        } else {
            Node rNode = new Node();
            // root 설정
            rNode.isRoot = true;
            rNode.isLeaf = false;

            // 새로운 루트가 생겨서 leaf 노드됨
            node.isRoot = false;
            newNode.isRoot = false;

            // 루트 노드 번호 설정 및 위치
            rNode.num = totalNode;
            rootLocation = rNode.num * blocksize;
            System.out.println("R node number: " + rNode.num);
            totalNode++;

            // 만든 루트노드에 값을 삽입
            System.out.println("여기 통과");
            System.out.println(newNode.p.get(0).first + " " + newNode.num);
            internalInsertion(rNode, newNode.p.get(0).first, newNode, node);

            newNode.isLeaf = node.isLeaf;

            // 루트 노드 정보 갱신
            data = convertObjectToBytes(rNode);
            writeToFile(data,  position: rNode.num * blocksize);
        }
        // 같은 레벨의 다음 노드를 표시 하기 위한 계산
        if (!node.isRoot) {
            // split 된 노드의 가장 마지막의 포인터는 기존의 node의 포인터가 됨
            newNode.r = node.r;
            // 기존 노드의 가장 마지막 포인터는 newNode를 가리킴
```

이를 해결하기 위해 각 기능을 함수해서 코드를 직관적으로 이해하기 쉽게 만들었고, 이에 따라 로직에서의 오류또한 조금 더 쉽게 접근할 수 있도록 했습니다.

```
public int Search(int key) throws IOException {
    if (this.isLeaf) {
        int pos = Arrays.binarySearch(this.keys, fromIndex: 0, this.nKeys, key);
        return pos >= 0 ? this.values[pos] : -1;
    } else {
        Node node = this.GetChild(key, isInsert: 0);
        return node.Search(key);
    }
}
```

```
public int Split() throws IOException {
    ++HanyangSEBPlusTree.this.nodecnt;
    Node newNode = HanyangSEBPlusTree.this.new Node(this.isLeaf, pos: -1);
    int mid = this.nKeys / 2;
    int end = this.nKeys;
    int leafInt = this.isLeaf ? 0 : 1;

    int idx;
    for(idx = 0; idx < end - mid - leafInt; ++idx) {
        newNode.keys[idx] = this.keys[idx + mid + leafInt];
    }

    newNode.nKeys = end - (mid + leafInt);
    this.nKeys -= end - (mid + leafInt);

    for(idx = 0; idx < end - mid; ++idx) {
        newNode.values[idx] = this.values[idx + mid + leafInt];
    }

    HanyangSEBPlusTree.this.cache.SetNode(newNode.pos, newNode);
    return this.keys[mid];
}
```

```
public void InsertValue(int key, int value) throws IOException {
    int valueIndex;
    int idx;
    Node newRootNode;
    if (this.isLeaf) {
        int pos = Arrays.binarySearch(this.keys, fromIndex: 0, this.nKeys, key);
        valueIndex = pos >= 0 ? pos : -pos - 1;

        for(idx = this.nKeys - 1; idx >= valueIndex; --idx) {
            this.keys[idx + 1] = this.keys[idx];
            this.values[idx + 1] = this.values[idx];
        }

        ++this.nKeys;
        this.keys[valueIndex] = key;
        this.values[valueIndex] = value;
        HanyangSEBPlusTree.this.cache.SetNode(this.pos, this);
    } else {
        newRootNode = this.GetChild(key, isInsert: 1);
        newRootNode.InsertValue(key, value);
        if (newRootNode.nodeIsFull()) {
            valueIndex = newRootNode.Split();
            idx = HanyangSEBPlusTree.this.nodecnt * HanyangSEBPlusTree.this.blockSize;
            this.InsertNode(valueIndex, idx);
            HanyangSEBPlusTree.this.cache.SetNode(this.pos, this);
        }
    }

    if (HanyangSEBPlusTree.this.root.nodeIsFull()) {
        ++HanyangSEBPlusTree.this.nodecnt;
        newRootNode = HanyangSEBPlusTree.this.new Node(isLeaf: false, pos: -1);
        newRootNode.keys[0] = this.Split();
        ++newRootNode.nKeys;
        newRootNode.values[0] = this.pos;
        newRootNode.values[1] = HanyangSEBPlusTree.this.nodecnt * HanyangSEBPlusTree.this.blockSize;
        HanyangSEBPlusTree.this.root = newRootNode;
    }
```

C. 처음 코드를 이해하고 bplus tree 코드를 구성하면서 nblock을 어떻게 사용해야 할지 감이 잘 오지 않았습니다. bplus tree의 효율도 높이고 싶었습니다.

이를 위해 교수님과 조교님께 질문 드렸습니다. 답변을 통해 nblock은 많은 노드 정보를 메모리에 한 번에 들고 있지 않고, 메모리를 효율적으로 사용하기 위해 주어진 변수라는 것을 알게 되었습니다. 또한 속도를 높이기 위해서는 위의 특성을 이용한 cache 기능을 사용해야 한다는 것을 알게 되었습니다.

cache를 구현하기 위해 조사를 했고 다음과 같은 알고리즘으로 cache를 구현할 수 있었습니다.

LRU(Least Recently Used Algorithm)

가장 오랫동안 참조되지 않은 값을 삭제하는 기법

메모리를 효율적으로 관리하기 위해 가장 최근에 사용된 데이터를 앞에 할당하고 가장 오래동안 쓰이지 않은 데이터를 뒤에 할당합니다.

이후 가용 메모리 보다 데이터가 커질 때 오랫동안 사용하지 않은 데이터부터 교체(저희 bplus 구현에서는 file 쓰기)합니다.

시간	1	2	3	4	5	6
참조 스트링	**1**	**2**	**3**	**1**	**4**	**5**
주기억장치 상태	1	1	1	2	2	3
		2	2	3	3	1
			3	1	1	4
					4	5

이렇게 함으로써 가용 메모리를 효율적으로 쓸 수 있습니다. 또한 데이터를 file에 직접적으로 읽고 쓰는 시간이 줄어들어 bplus tree의 속도를 향상시킬 수 있었습니다.

코드상에서는 새 노드가 생기면 cache에 할당합니다. cache 안의 노드 정보가 변환될 경우에는 정보를 재설정합니다. 또한 노드를 찾아야 할 경우에도 cache에 접근해 빠르게 찾을 수 있습니다.

```
cache.SetNode(newNode.pos, newNode);//캐시에 새 노드를 설정
return keys[mid];//분할된 첫 번째 키를 반환 -> 상위 노드에 삽입되며, 분할되는 노드들을 분리하는 역할
```

```
cache.SetNode(this.pos, this);//캐시에 해당 노드 설정
```

```
Node GetChild(int key, int isInsert) throws IOException {

    int pos = Arrays.binarySearch(this.keys, fromIndex: 0, this.n
    int childIndex = pos >= 0 ? (pos + 1) : (-pos - 1);//삽입될
    int childValue = this.values[childIndex];//찾은 위치에 해당하는
    return cache.GetNode(childValue);//해당 노드를 캐시에서 끌고옴

}
```

이렇게 cache를 이용하면서 용량이 가득 차게 되면 가장 오래된 데이터 부터 file에 저장합니다.

```
new *
@Override
protected boolean removeEldestEntry(Map.Entry<Integer, Data> eldest) {//오래된 항목 제거
    if (size() == capacity)//캐시용량이 가득차면
        try {
            LRU.this.RemoveNode(eldest);
        } catch (IOException e) {

        }
    return size() > LRU.this.capacity;//캐시용량 초과여부 반환
}
```

실제로 위 알고리즘을 사용해 cache를 사용함으로써 bplus tree의 속도를 평균 6~7초대로 속도를 개선할 수 있었습니다.

3. 느낀점

처음 사용해보는 class나 기법, 자료구조 등을 사용해 꽤 복잡한 논리구조를 구현해야 해서 시작할 때는 막막하기만 했었다. 하지만 팀원분들과 업무를 나누고, 지식을 공유하고, 새로운 정보를 알아 이를 사용해 보는 과정에서 업무의 흐름과 새로운 기법이나, 기술을 어떻게 공부하고 사용해야 하는지 조금이나마 알 수 있었다.

메소드들이 너무 많아지다 보니 에러가 발생해도 어디서 에러가 발생했는지 디버깅도 힘들고 한 곳을 고치면 다른 곳이 에러가 나는 등 진전이 없었는데 전체적인 흐름을 최대한 간단하고 보기 편하게 구현하려고 노력했더니 훨씬 디버깅도 편하고 이해도도 올라간 것 같았습니다.