데이터베이스 설계 Part1

1. 기능 테스트 데이터

A. 블록 설정

i. 모든 데이터 이동은 일정 크기(BLOCK_SIZE) 단위로 수행된다.

ii. BLOCK_SIZE는 상수로 정의되어 필요할 시 변경이 가능하도록 한다.

B. 레코드 및 헤더 블록 구성

i. 파일은 첫 번째 블록을 헤더 블록으로 하며, 이 헤더에는 파일 내 첫 레코드의 포인터와 메타데이터(필드 개수, 필드 명칭, 각 필드의 바이트 길이, 필드 순서 등)가 기록된다.

ii. 레코드 블록에는 각 레코드가 (포인터 + null 비트맵 + 고정 길이 필드 데이터)의 구조로 저장된다.

C. 테스트 데이터

i. 테스트는 편의성을 위해 블록과 레코드의 크기를 작게 설정하여 한 파일에 총 10개의 레코드를 삽입하도록 구성한다.

ii. 각 레코드는 3개의 필드로 구성되며, 탐색 키인 첫 번째 필드(A)는 4바이트, 나머지 필드(B, C)는 각각 3바이트로 구성한다.

1. A: int(4) / nullable
2. B: char(3) / nullable
3. C: char(3) / nullable

iii. 예시 데이터 파일의 내용은 다음과 같다.

```
f1,10,0001;X01;Y01|0002;null;Y02|0003;X03;null|0004;null;Y04|0005;X05;null|0006;X06;Y06|0007;null;null|0008; null;Y08|0009;X09;null|0010;X10;Y10
```

A	B	C
0001	X01	Y01
0002	null	Y02
0003	X03	null
0004	null	Y04
0005	X05	null
0006	X06	Y06
0007	null	null
0008	null	Y08
0009	X09	null
0010	X10	Y10

2. 설계

A. 파일 구조 및 블록 단위 I/O

i. 헤더 블록

1. 파일의 첫 번째 블록은 헤더 블록으로 구성되며, BLOCK_SIZE와 동일한 크기를 가진다.
2. 헤더에는 파일의 첫 레코드 위치인 포인터와 함께 메타데이터인 필드 개수, 각 필드의 이름, 바이트 길이, 필드 순서, 탐색키 정보 등이 저장된다.

ii. 레코드 저장 방식

1. 각 레코드는 블록 단위로 기록되며, 기본 구조는 다음과 같다.
   A. 포인터 필드: 해당 레코드 이후 다음 레코드의 오프셋을 저장하는 필드.
   B. Null 비트맵: 탐색키를 제외한 나머지 필드의 null 여부
   C. 필드 데이터: 고정 길이 문자열 형태로 저장되고 각 필드의 크기는 헤더에 기록된 바이트 길이를 따름
2. 레코드 삽입 시 탐색 키인 첫 번째 필드를 기준으로 순차 정렬을 유지하도록 Linked List 방식의 포인터 업데이트를 수행한다.

iii. 블록 크기 및 설정 변경

1. 모든 I/O 연산은 BLOCK_SIZE 상수에 따라 이루어진다.
2. 추후 테스트 시에는 블록과 레코드 크기를 작게 설정하여 입출력 결과를 용이하게 확인할 수 있도록 한다.

B. 레코드 삽입 및 검증

i. 삽입 기능

1. 레코드 삽입은 레코드 삽입용 데이터를 한 줄 단위로 읽고, 각 레코드를 파싱하여 순차적으로 삽입하는 방식으로 구현된다.
2. 데이터 파일의 한 줄은 '파일명, 레코드 수, 레코드 데이터'의 형식으로 구성되고, 레코드 데이터는 파이프(|)로 구분하며, 각 레코드는 세미콜론(;)으로 필드를 구분한다.
3. 삽입 시 null 비트맵을 계산하여 각 레코드의 null 처리 여부를 기록하고 삽입 후 자동으로 전체 범위 탐색 키 검색을 수행하여 작업이 정확히 수행되었는지를 검증한다.

ii. 자동 검증 기능: 레코드 삽입 후 사전에 지정된 탐색 키 범위(전체 범위)를 자동으로 검색하고 결과를 출력하여 기능이 정확하게 동작하고 있음을 사용자에게 보여준다.

C. 필드 및 레코드 검색

i. 필드 검색

1. 사용자가 파일명과 검색할 필드명을 지정하면 해당 파일 내 모든 레코드에서 지정 필드의 값을 추출하여 출력한다.
2. 헤더에 저장된 메타데이터를 기반으로 각 레코드에서 특정 필드의 위치를 계산하여 데이터를 읽고, null 비트맵이 존재할 시 'null'로 처리할 수 있도록 설계한다.

ii. 레코드 범위 검색

1. 첫 번째 필드인 탐색 키의 값을 기준으로 사용자가 지정한 범위(minKey <= A <= maxKey)에 포함되는 레코드를 검색하여 출력한다.

2. 이 기능은 모든 레코드를 대상으로 할 수 있어야 하며, 범위 검색 결과는 삽입 후 자동 검증 기능으로서 적용되어야 한다.

D. 메타데이터 관리

i. 메타데이터 저장 방법

1. MySQL의 JDBC를 활용하여 디스크 상 파일과 별도로 메타데이터를 관계테이블에 저장하는 방식을 사용한다.
2. 메타데이터 테이블은 파일명, 필드 순서, 필드 명, 필드 길이 정보를 저장한다.
3. 파일 생성 시 메타데이터는 MySQL DB에도 저장되어 추후 해당 파일의 구조를 확인하거나 분석할 수 있도록 한다.

ii. 메타데이터 조회: JDBC를 통해 저장된 메타데이터 정보를 조회함으로써 프로그램이 실행될 때 파일의 구조를 동적으로 파악하고 레코드 삽입 및 검색 작업을 지원한다.

3. 구현

A. 설계 내용 중 변경 사항

i. 레코드 삽입 및 포인터 업데이트

1. 각 필드의 명칭의 글자 수가 2의 배수가 되도록 A0, B0, C0로 변경하며, 필드의 크기도 전부 4바이트로 통일한다.
2. 마지막 레코드를 찾은 뒤 마지막 레코드가 새 레코드를 가리켜 포인터를 갱신하도록 한다.
3. 레코드를 블록 단위로 딱 맞춰 저장하고 파일 길이는 항상 블록 크기*n의 형태로 증가하도록 한다.

ii. 블록 구조

1. 디스크 상의 데이터를 고정 크기의 블록 단위로 처리하며, 각 블록은 파일 구조 내에서 레코드를 저장하기 위한 최소단위다. 블록 크기는 4096 바이트로 설정되어 있고 그 구조는 아래의 두 종류로 나뉜다.
2. 헤더 블록: 파일의 첫 번째 블록이고 전체 파일의 메타데이터를 저장한다.
3. 데이터 블록: 실제 레코드들을 저장하는 블록이며 다수의 레코드들을 순차적으로 저장한다. 새로운 레코드는 가장 마지막 레코드의 다음 포인터를 갱신한 뒤 이어 붙이는 방식으로 연결된다.
4. 블록 내에 레코드를 삽입할 공간이 부족할 경우 자동으로 다음 블록으로 넘어가 새로운 레코드를 저장한다.

iii. null 필드 처리 방식

1. 이전 설계에서는 null 처리를 어떻게 할 지 구체적으로 명시되지 않았다.
2. 레코드 삽입용 데이터 파일에서 읽어온 필드가 null이거나 빈 문자열일 경우 null 필드로 표시하고 실제 저장 값은 비어 있도록 한다. 파일을 읽을 때 해당 필드는 저장 공간이 없고 검색 시 null로 표시된다.

iv. 자동 검색 범위

1. 삽입 후 자동 검증을 위해 전체 레코드를 확인하도록 한다.

v. 필드 및 레코드 검색

1. 레코드를 순회할 때 이전 레코드에서의 다음 레코드 블록과 오프셋을 따라가며 파일 끝까지 검색한다. 이때 null 필드인지 아니면 실제 값이 있는지 판별한다.

vi. 메타데이터 저장

1. 헤더 블록에 기본 메타데이터를 저장하도록 설계하였으므로 메모리에 로딩된 메타데이터 객체를 활용한다.

B. 기능별 구현 로직

i. FileStructure.java

1. class FieldInfo: 파일의 각 필드에 대한 정보를 생성한다.

```
13    public static class FieldInfo {
14        String name;
15        int length;
16        public FieldInfo(String name, int length) {
17            this.name = name;
18            this.length = length;
19        }
20    }
```

2. class Metadata: 파일의 헤더 블록에 기록될 메타데이터(첫 레코드 위치, 필드 개

수, 각 필드 구조 등)를 생성한다.

```
22    public static class Metadata {
23        int firstRecordBlock;
24        int firstRecordOffset;
25        int fieldCount;
26        List<FieldInfo> fields;
27
28        public Metadata(List<FieldInfo> fields) {
29            this.firstRecordBlock = -1;
30            this.firstRecordOffset = -1;
31            this.fieldCount = fields.size();
32            this.fields = fields;
33        }
34    }
```

- firstRecordBlock, firstRecordOffset: 파일에서 첫 레코드가 위치하는 블록 번호와 블록 오프셋으로, 헤드 블록에 저장되어 전체 레코드를 탐색하는 시작점이 된다.
- fieldCount: 필드(컬럼)의 개수
- fields: 각 필드의 이름과 길이를 저장하는 리스트

3. class Record: 실제 레코드를 나타내며 연결 리스트 구조로 다음 레코드로 이동 가능하도록 설계한다.

```
36    public static class Record {
37        int nextRecordBlock;
38        int nextRecordOffset;
39        List<String> fieldValues;
40        byte nullBitmap;
41
42        public Record(List<String> fieldValues, byte nullBitmap) {
43            this.nextRecordBlock = -1;
44            this.nextRecordOffset = -1;
45            this.fieldValues = fieldValues;
46            this.nullBitmap = nullBitmap;
47        }
48    }
```

- nextRecordBlock, nextRecordOffset: 연결 리스트처럼 다음 레코드가 위치한 블록 번호와 오프셋을 기록한다.
- fieldValues: 실제 필드 문자열 값들의 리스트
- nullBitmap: 특정 필드의 null 여부를 비트로 표시하며, 해당 비트가 1이면 null로 간주한다.

4. writeHeader(): 파일의 맨 앞인 헤더 블록에 Metadata 정보를 BLOCK_SIZE(4096바이트) 단위로 기록한다.

```
50    public void writeHeader(RandomAccessFile raf, Metadata meta) throws IOException {
51        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
52        buffer.putInt(meta.firstRecordBlock);
53        buffer.putInt(meta.firstRecordOffset);
54        buffer.putInt(meta.fieldCount);
55
56        for (FieldInfo field : meta.fields) {
57            byte[] nameBytes = field.name.getBytes(StandardCharsets.UTF_8);
58            byte[] fixedName = new byte[FIELD_NAME_SIZE];
59            System.arraycopy(nameBytes, 0, fixedName, 0, Math.min(nameBytes.length, FIELD_NAME_SIZE));
60            buffer.put(fixedName);
61            buffer.putInt(field.length);
62        }
63
64        raf.seek(0);
65        raf.write(buffer.array());
66    }
```

- ByteBuffer를 BLOCK_SIZE만큼 할당한다.
- 메타데이터의 firstRecordBlock, firstRecordOffset, fieldCount를 기록한다.
- 각 필드의 이름(고정 길이 20바이트)과 길이를 순차적으로 버퍼에 채운다.
- 파일 포인터를 0으로 이동(raf.seek(0))하여, 버퍼 전체를 디스크에 기록한다.

5. readHeader(): 파일 맨 앞에서 헤더 블록을 읽어 Metadata 객체를 생성한다.

```java
public Metadata readHeader(RandomAccessFile raf) throws IOException {
    ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
    raf.seek(0);
    raf.readFully(buffer.array());
    buffer.rewind();
    int firstRecordBlock = buffer.getInt();
    int firstRecordOffset = buffer.getInt();
    int fieldCount = buffer.getInt();
    List<FieldInfo> fields = new ArrayList<>();

    for (int i = 0; i < fieldCount; i++) {
        byte[] fixedName = new byte[FIELD_NAME_SIZE];
        buffer.get(fixedName);
        String name = new String(fixedName, StandardCharsets.UTF_8).trim();
        int length = buffer.getInt();
        fields.add(new FieldInfo(name, length));
    }

    Metadata meta = new Metadata(fields);
    meta.firstRecordBlock = firstRecordBlock;
    meta.firstRecordOffset = firstRecordOffset;
    return meta;
}
```

- BLOCK_SIZE 크기의 버퍼를 준비하고 파일 포인터를 0으로 이동.
- 헤더 블록(4096바이트)을 읽어들여 ByteBuffer에 넣는다.
- 헤더에 포함된 firstRecordBlock, firstRecordOffset, fieldCount를 차례로 읽는다.
- 필드 개수만큼 길이를 읽어 FieldInfo 리스트를 구성한다.
- Metadata 객체를 만들어 반환한다.

6. getRecordSize(): 레코드 크기 계산

```java
private int getRecordSize(Metadata meta) {
    int size = 4 + 4 + 1; // nextBlock + nextOffset + nullBitmap
    for (FieldInfo field : meta.fields) {
        size += field.length;
    }
    return size;
}
```

7. insertRecord(): 파일에 새 레코드를 삽입하고 연결 리스트 구조를 갱신한다.

```java
public void insertRecord(String filename, Metadata meta, Record newRecord) throws IOException {
    try (RandomAccessFile raf = new RandomAccessFile(filename, "rw")) {
        int recordSize = getRecordSize(meta);
        int fileLength = (int) raf.length();
        int lastBlockStart = (fileLength / BLOCK_SIZE) * BLOCK_SIZE;
        int offsetInLastBlock = fileLength % BLOCK_SIZE;

        int newBlock = fileLength / BLOCK_SIZE;
        int newOffset = offsetInLastBlock;

        if (offsetInLastBlock + recordSize > BLOCK_SIZE) {
            newBlock++;
            newOffset = 0;
        }

        if (meta.firstRecordBlock == -1 && meta.firstRecordOffset == -1) {
            meta.firstRecordBlock = newBlock;
            meta.firstRecordOffset = newOffset;
            raf.seek(0);
            raf.writeInt(meta.firstRecordBlock);
            raf.writeInt(meta.firstRecordOffset);
        } else {
            int currBlock = meta.firstRecordBlock;
            int currOffset = meta.firstRecordOffset;

            while (true) {
                raf.seek((long) currBlock * BLOCK_SIZE + currOffset);
                int nextBlock = raf.readInt();
                int nextOffset = raf.readInt();

                if (nextBlock == -1 && nextOffset == -1) {
                    raf.seek((long) currBlock * BLOCK_SIZE + currOffset);
                    raf.writeInt(newBlock);
                    raf.writeInt(newOffset);
```

```
                    break;
                } else {
                    currBlock = nextBlock;
                    currOffset = nextOffset;
                }
            }
        }

        writeRecord(raf, newBlock, newOffset, meta, newRecord);
    }
}
```

- 파일 길이를 구해 새 레코드를 쓸 위치(블록 번호와 오프셋) 계산한다.
- 만약 파일에 레코드가 없다면(헤더 정보가 -1, -1)
  1. 헤더의 firstRecordBlock, firstRecordOffset를 현재 레코드 위치로 갱신한다.
  2. 디스크의 맨 앞(헤더 위치)로 이동하여 해당 정보를 다시 기록한다.
- 이미 레코드가 존재한다면
  1. 헤더에 기록된 첫 레코드로부터 nextRecordBlock, nextRecordOffset를 따라가며 연결 리스트의 마지막 레코드를 찾는다.
  2. 마지막 레코드의 포인터를 새 레코드 위치로 갱신한다.
- writeRecord() 메소드를 호출하여 새 레코드를 실제 디스크에 기록.

8. writeRecord(): 지정된 블록 번호와 오프셋 위치에, 하나의 레코드를 BLOCK_SIZE 단위로 실제 기록한다.

```
public void writeRecord(RandomAccessFile raf, int block, int offset, Metadata meta, Record record) throws IOException {
    int recordSize = getRecordSize(meta);
    raf.seek((long) block * BLOCK_SIZE + offset);
    ByteBuffer buffer = ByteBuffer.allocate(recordSize);

    buffer.putInt(record.nextRecordBlock);
    buffer.putInt(record.nextRecordOffset);
    buffer.put(record.nullBitmap);

    for (int i = 0; i < meta.fieldCount; i++) {
        if (((record.nullBitmap >> i) & 1) == 0) {
            String value = record.fieldValues.get(i);
            byte[] valueBytes = value.getBytes(StandardCharsets.UTF_8);
            byte[] fixedField = new byte[meta.fields.get(i).length];
            Arrays.fill(fixedField, (byte) ' ');
            System.arraycopy(valueBytes, 0, fixedField, 0, Math.min(valueBytes.length, fixedField.length));
            buffer.put(fixedField);
        }
    }

    raf.write(buffer.array());
}
```

- 파일 포인터를 (block * BLOCK_SIZE + offset)로 이동한다.
- BLOCK_SIZE 크기의 ByteBuffer를 만들고 다음 정보를 순서대로 기록한다.
  1. nextRecordBlock와 nextRecordOffset(각 4바이트)
  2. nullBitmap(1바이트)
  3. 각 필드 데이터(고정 길이): 만약 nullBitmap에서 해당 필드가 null이라면(비트=1) 데이터를 쓰지 않는다. null이 아니라면 필드 길이만큼의 공간에 값을 작성한다.
- 기록 완료 시 이 버퍼를 raf.write()로 디스크에 쓴다.

ii. RecordBulkInserter.java

1. bulkInsertFromDataFile(): 레코드 삽입용 데이터 파일을 읽어와 그 안에 정의된 데이터들을 .dat 파일에 삽입한다.

```java
public void bulkInsertFromDataFile(String fileName, String dataFilePath) {
    try (BufferedReader br = new BufferedReader(new FileReader(dataFilePath))) {
        String dataFilename = fileName + ".dat";

        FileStructure.Metadata meta;
        try (RandomAccessFile raf = new RandomAccessFile(dataFilename, "rw")) {
            meta = fileStructure.readHeader(raf);
        } catch (IOException e) {
            System.out.println("Failed to read header from file: " + fileName);
            return;
        }

        List<FileStructure.Record> recordList = new ArrayList<>();
        List<String> searchKeys = new ArrayList<>();

        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;

            String[] parts = line.split(",", 2);
            if (parts.length < 2) {
                System.out.println("Invalid record line: " + line);
                continue;
            }

            String recordsData = parts[1].trim();
            String[] recordStrArr = recordsData.split("\\|");
            for (String recStr : recordStrArr) {
                String[] fieldValuesArr = recStr.split(";");
                List<String> fieldValues = new ArrayList<>();
                byte nullBitmap = 0;

                for (int i = 0; i < fieldValuesArr.length; i++) {
                    String val = fieldValuesArr[i].trim();
                    if (val.isEmpty() || val.equalsIgnoreCase("null")) {
                        nullBitmap |= (1 << i);
                        fieldValues.add("");
                    } else {
                        fieldValues.add(val);
                    }
                }

                String searchKey = fieldValues.get(0);
                if (!searchKey.isEmpty()) {
                    searchKeys.add(searchKey);
                }

                FileStructure.Record record = new FileStructure.Record(fieldValues, nullBitmap);
                recordList.add(record);
            }
        }

        recordList.sort((r1, r2) -> r1.fieldValues.get(0).compareTo(r2.fieldValues.get(0)));

        for (int i = 0; i < recordList.size() - 1; i++) {
            String currKey = recordList.get(i).fieldValues.get(0);
            String nextKey = recordList.get(i + 1).fieldValues.get(0);
            if (currKey.equals(nextKey)) {
                System.out.println("Insertion failed due to duplicated keys in record file.");
                return;
            }
        }

        for (FileStructure.Record rec : recordList) {
            fileStructure.insertRecord(dataFilename, meta, rec);
        }

        System.out.println("Records inserted into file '" + dataFilename + "' successfully (no duplicate keys).");

        metadataManager.createTableInMySQL(fileName, meta);
        insertAllRecordsIntoMySQL(fileName, meta, recordList);

        if (!searchKeys.isEmpty()) {
            Collections.sort(searchKeys);
            String minKey = searchKeys.get(0);
            String maxKey = searchKeys.get(searchKeys.size() - 1);
            System.out.println("Auto search: range " + minKey + " ~ " + maxKey);
            recordSearcher.searchRecordsByKeyRange(fileName, minKey, maxKey);
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- 지정된 .dat 파일을 열고 메타데이터를 읽어 필드 정보 확보한다.
- 입력 줄에서 레코드 문자열을 추출하며, 개별 레코드를 분리하고 필드를 나눈다.
- 탐색키 중복 검사: 삽입 전 모든 탐색키를 Set에 넣어 중복이 있는지 확인하고, 중복이 있으면 작업을 중복한다.
- 레코드 정렬: 필드 리스트를 탐색키 기준으로 오름차순 정렬한다.
- 정렬된 레코드에서 null 처리 비트맵 설정 후 .dat 파일에 삽입한다.
- 삽입 완료 후 탐색 키의 전체 범위에서 레코드들을 출력한다.

iii. FieldSearcher.java

1. readHeader(): 파일의 헤더 블록을 읽어 해당 파일이 가진 메타데이터를 복원한

다.

```
    public FileStructure.Metadata readHeader(RandomAccessFile raf) throws IOException {
        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
        raf.seek(0);
        raf.readFully(buffer.array());
        buffer.rewind();
        int firstRecordBlock = buffer.getInt();
        int firstRecordOffset = buffer.getInt();
        int fieldCount = buffer.getInt();

        FileStructure.Metadata meta = new FileStructure.Metadata(new java.util.ArrayList<>());
        meta.firstRecordBlock = firstRecordBlock;
        meta.firstRecordOffset = firstRecordOffset;
        meta.fieldCount = fieldCount;

        for (int i = 0; i < fieldCount; i++) {
            byte[] fixedName = new byte[FIELD_NAME_SIZE];
            buffer.get(fixedName);
            String name = new String(fixedName, StandardCharsets.UTF_8).trim();
            int length = buffer.getInt();
            meta.fields.add(new FileStructure.FieldInfo(name, length));
        }
        return meta;
    }
```

- 헤더 블록 읽기
    1. 블록 크기(BLOCK_SIZE = 4096) 만큼 버퍼를 할당한다.
    2. seek(0)으로 파일의 첫 부분인 헤더 블록으로 이동한다.
    3. 전체 4096바이트를 읽어 ByteBuffer에 로드한다.
- 메타데이터 파싱: 첫 레코드의 위치와 필드 개수를 읽는다.
- 필드 정보 파싱: 필드 개수만큼 반복하면서 각 필드명과 길이를 읽어 FieldInfo로 만들어 저장한다.
- 메타데이터 객체 생성: 읽은 값들로 Metadata를 구성하고 반환한다.

2. searchField()

```
    public void searchField(String fileName, String searchField) {
        String dataFilename = fileName + ".dat";
        try (RandomAccessFile raf = new RandomAccessFile(dataFilename, "r")) {
            FileStructure.Metadata meta = readHeader(raf);

            int fieldIndex = -1;
            for (int i = 0; i < meta.fieldCount; i++) {
                if (meta.fields.get(i).name.equalsIgnoreCase(searchField)) {
                    fieldIndex = i;
                    break;
                }
            }
            if (fieldIndex == -1) {
                System.out.println("Cannot find field '" + searchField + "'.");
                return;
            }

            int block = meta.firstRecordBlock;
            int offset = meta.firstRecordOffset;

            while (block != -1 && offset != -1) {
                raf.seek((long) block * BLOCK_SIZE + offset);

                int nextRecordBlock = raf.readInt();
                int nextRecordOffset = raf.readInt();
                byte nullBitmap = raf.readByte();

                int dataOffset = 9; // pointer(8byte) + nullBitmap(1byte)
                for (int i = 0; i < fieldIndex; i++) {
                    if (((nullBitmap >> i) & 1) == 0) {
                        dataOffset += meta.fields.get(i).length;
                    }
                }

                if (((nullBitmap >> fieldIndex) & 1) == 0) {
                    raf.seek((long) block * BLOCK_SIZE + offset + dataOffset);
                    byte[] fieldBytes = new byte[meta.fields.get(fieldIndex).length];
                    raf.read(fieldBytes);
                    String fieldValue = new String(fieldBytes, StandardCharsets.UTF_8).trim();
                    System.out.println("Block:" + block + " Offset:" + offset + " Field " + searchField + " Value: " + fieldValue);
                } else {
                    System.out.println("Block:" + block + " Offset:" + offset + " Field " + searchField + " Value: null");
                }

                block = nextRecordBlock;
                offset = nextRecordOffset;
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

- 헤더 읽기: 헤더 정보로 전체 필드 목록 및 firstRecordBlock과 firstRecordOffset을 가져온다.

- 필드 인덱스 검색: 원하는 필드명이 몇 번째 인덱스인지 탐색한다.
- 레코드 순회
    1. 헤더에 기록된 firstRecordBlock, firstRecordOffset부터 시작하여 레코드를 연결 리스트 형태로 추적한다.
    2. 각 레코드에 대해 포인터 및 nullBitmap을 읽고 다음 레코드로 이동한다.
- 해당 필드 데이터 추출: 레코드 내에서 탐색하고자 하는 필드의 시작 지점을 계산한다. 이때 null 비트맵에서 해당 필드가 null이면 실제 데이터 길이를 추가하지 않는다.
- 값 출력: 고정 길이만큼의 바이트를 일어 문자열로 변환한 후 출력한다. 이때 해당 필드가 null이면 null을 출력한다.

iv. RecordSearcher.java

1. readHeader(): .dat 파일의 헤더 블록을 읽고 메타데이터를 복원한다. 구조는 FieldSearcher의 readHeader()와 동일하다.

```
11    public FileStructure.Metadata readHeader(RandomAccessFile raf) throws IOException {
12        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
13        raf.seek(0);
14        raf.readFully(buffer.array());
15        buffer.rewind();
16        int firstRecordBlock = buffer.getInt();
17        int firstRecordOffset = buffer.getInt();
18        int fieldCount = buffer.getInt();
19
20        FileStructure.Metadata meta = new FileStructure.Metadata(new java.util.ArrayList<>());
21        meta.firstRecordBlock = firstRecordBlock;
22        meta.firstRecordOffset = firstRecordOffset;
23        meta.fieldCount = fieldCount;
24
25        for (int i = 0; i < fieldCount; i++) {
26            byte[] fixedName = new byte[FIELD_NAME_SIZE];
27            buffer.get(fixedName);
28            String name = new String(fixedName, StandardCharsets.UTF_8).trim();
29            int length = buffer.getInt();
30            meta.fields.add(new FileStructure.FieldInfo(name, length));
31        }
32        return meta;
33    }
```

2. searchRecordsByKeyRange(): .dat 파일에서 탐색 키에 대해 minKey <= key <= maxKey인 모든 레코드를 찾아 정보를 출력한다.

```
35    public void searchRecordsByKeyRange(String fileName, String minKey, String maxKey) {
36        String dataFilename = fileName + ".dat";
37        try (RandomAccessFile raf = new RandomAccessFile(dataFilename, "r")) {
38            FileStructure.Metadata meta = readHeader(raf);
39            int keyLength = meta.fields.get(0).length;
40
41            int block = meta.firstRecordBlock;
42            int offset = meta.firstRecordOffset;
43
44            while (block != -1 && offset != -1) {
45                raf.seek((long) block * BLOCK_SIZE + offset);
46                int nextRecordBlock = raf.readInt();
47                int nextRecordOffset = raf.readInt();
48                byte nullBitmap = raf.readByte();
49
50                byte[] keyBytes = new byte[keyLength];
51                raf.read(keyBytes);
52                String key = new String(keyBytes, StandardCharsets.UTF_8).trim();
53
54                if (key.compareTo(minKey) >= 0 && key.compareTo(maxKey) <= 0) {
55                    System.out.println("Block:" + block + " Offset:" + offset + " Search-key:" + key);
56
57                    int dataOffset = 9 + keyLength;
58                    for (int i = 1; i < meta.fieldCount; i++) {
59                        if (((nullBitmap >> i) & 1) == 0) {
60                            raf.seek((long) block * BLOCK_SIZE + offset + dataOffset);
61                            byte[] fieldBytes = new byte[meta.fields.get(i).length];
62                            raf.read(fieldBytes);
```

```
                        String fieldValue = new String(fieldBytes, StandardCharsets.UTF_8).trim();
                        System.out.println(meta.fields.get(i).name + ": " + fieldValue);
                        dataOffset += meta.fields.get(i).length;
                    } else {
                        System.out.println(meta.fields.get(i).name + ": null");
                    }
                }
            }

            block = nextRecordBlock;
            offset = nextRecordOffset;
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- 헤더 읽기: 헤더를 읽어 파일의 첫 레코드 위치, 필드 구조, 첫 필드의 길이를 확인한다.
- 레코드 순회: 헤더에 기록된 첫 레코드의 블록과 오프셋부터 시작하여 각 레코드의 블록 및 오프셋을 따라가 연결 리스트를 탐색한다.
- 탐색 키 읽기 및 범위 비교: 레코드의 탐색 키를 문자열로 변환하고 정해진 범위 내에 있을 때만 해당 레코드 정보를 출력한다.
- 나머지 정보 출력: 두 번째 필드부터 순회하며 특정 필드가 null이 아니면 고정 길이만큼 읽어 문자열을 반환한다.
- 포인터 이동: 현재 레코드의 블록 및 오프셋을 다음 레코드로 루프 한다.

v. MetadataManager.java

1. loadMetadataFromFile(): 지정된 파일에서 헤더 블록을 읽고 그 안에 저장된 필드 정보를 로드한다.

```
public void loadMetadataFromFile(String fileName) {
    String dataFile = fileName + ".dat";
    try (RandomAccessFile raf = new RandomAccessFile(dataFile, "r")) {
        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
        raf.seek(0);
        raf.readFully(buffer.array());
        buffer.rewind();

        int firstRecordBlock = buffer.getInt();
        int firstRecordOffset = buffer.getInt();
        int fieldCount = buffer.getInt();

        List<FileStructure.FieldInfo> tempFields = new ArrayList<>();
        for (int i = 0; i < fieldCount; i++) {
            byte[] fixedName = new byte[FIELD_NAME_SIZE];
            buffer.get(fixedName);
            String name = new String(fixedName, StandardCharsets.UTF_8).trim();
            int length = buffer.getInt();
            tempFields.add(new FileStructure.FieldInfo(name, length));
        }

        fieldsInMemory = tempFields;
        System.out.println("Metadata is loaded to memory from File " + fileName + ".dat");
        System.out.println("Field numbers: " + fieldCount);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- 헤더 블록 읽기: 파일 맨 앞에서 BLOCK_SIZE=4096바이트만큼 읽어온다.
- 헤더의 기록된 첫 레코드 포인터와 필드 개수를 파싱하여 확인한다.
- 필드 정보 파싱: 각 필드의 이름과 길이를 읽어 결과 객체를 생성한 뒤 모은 결과를 리스트에 저장한다.
- 메모리 갱신: 읽어온 필드 정보들을 클래스 필드에 반영한다.

2. getFieldsInMemory(): 메모리에 현재 적재된 필드 목록을 반환한다.

```
public List<FileStructure.FieldInfo> getFieldsInMemory() {
    return fieldsInMemory;
}
```

- 다른 클래스에서 로딩된 메타데이터를 확인할 수 있게 한다.

3. saveMetadataToFile(): 현재 메모리에 있는 필드 목록을 다시 헤더 블록에 갱신한다.

```
51    public void saveMetadataToFile(String fileName, int firstRecordBlock, int firstRecordOffset) {
52        String dataFile = fileName + ".dat";
53        try (RandomAccessFile raf = new RandomAccessFile(dataFile, "rw")) {
54            ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
55
56            buffer.putInt(firstRecordBlock);
57            buffer.putInt(firstRecordOffset);
58            buffer.putInt(fieldsInMemory.size());
59
60            for (FileStructure.FieldInfo field : fieldsInMemory) {
61                byte[] nameBytes = field.name.getBytes(StandardCharsets.UTF_8);
62                byte[] fixedName = new byte[FIELD_NAME_SIZE];
63                System.arraycopy(nameBytes, 0, fixedName, 0, Math.min(nameBytes.length, FIELD_NAME_SIZE));
64                buffer.put(fixedName);
65                buffer.putInt(field.length);
66            }
67
68            raf.seek(0);
69            raf.write(buffer.array());
70            System.out.println("File " + fileName + ".dat's header information has been updated.");
71
72        } catch (IOException e) {
73            e.printStackTrace();
74        }
75    }
```

- 버퍼 초기화: 기록할 정보들을 버퍼에 순서대로 넣는다.
- 필드 정보 쓰기: 각 필드의 이름과 길이를 헤더에 기록한다.
- 헤더 업데이트: 파일의 맨 앞부터 4096바이트를 다시 써서 헤더 정보를 업데이트한다.

vi. MainApp.java

1. main()

```
7     public static void main(String[] args) {
8         Scanner scanner = new Scanner(System.in);
9         FileStructure fileStructure = new FileStructure();
10        RecordBulkInserter inserter = new RecordBulkInserter();
11        FieldSearcher fieldSearcher = new FieldSearcher();
12        RecordSearcher recordSearcher = new RecordSearcher();
13        MetadataManager metadataManager = new MetadataManager();
14
15        while (true) {
16            System.out.println("\n==== Database System ====");
17            System.out.println("1. Create new data file (from config .txt)");
18            System.out.println("2. Record Insertion (With data file)");
19            System.out.println("3. Search Field");
20            System.out.println("4. Search Record Range");
21            System.out.println("5. Check Metadata(Memory)");
22            System.out.println("6. Exit");
23            System.out.print("Selection: ");
24            String choice = scanner.nextLine();
25
26            switch (choice) {
27                case "1":
28                    System.out.print("Enter config text file path to create .dat: ");
29                    String configPath = scanner.nextLine().trim();
30                    boolean createSuccess = createNewDataFileFromConfig(configPath, fileStructure, metadataManager);
31                    if (!createSuccess) {
32                        System.out.println("Failed to create data file due to invalid config.");
33                    }
34                    break;
35
36                case "2":
37                    System.out.print("Enter file name to insert into (without extension): ");
38                    String insertFileName = scanner.nextLine().trim();
39                    System.out.print("Record insertion data file path: ");
40                    String dataFilePath = scanner.nextLine();
41                    inserter.bulkInsertFromDataFile(insertFileName, dataFilePath);
42                    break;
```

```
                case "3":
                    System.out.print("File name: ");
                    String fNameField = scanner.nextLine();
                    System.out.print("Search field name: ");
                    String field = scanner.nextLine();
                    fieldSearcher.searchField(fNameField, field);
                    break;

                case "4":
                    System.out.print("File name: ");
                    String fNameRange = scanner.nextLine();
                    System.out.print("Search-key minimum value: ");
                    String minKey = scanner.nextLine();
                    System.out.print("Search-key maximum value: ");
                    String maxKey = scanner.nextLine();
                    recordSearcher.searchRecordsByKeyRange(fNameRange, minKey, maxKey);
                    break;

                case "5":
                    System.out.print("Enter file name (without extension) to check metadata: ");
                    String metaFileName = scanner.nextLine().trim();
                    metadataManager.loadMetadataFromFile(metaFileName);

                    List<FileStructure.FieldInfo> loadedFields = metadataManager.getFieldsInMemory();
                    if (loadedFields.isEmpty()) {
                        System.out.println("No metadata loaded or file not found.");
                    } else {
                        System.out.println("List of fields in memory for file '" + metaFileName + ".dat':");
                        for (int i = 0; i < loadedFields.size(); i++) {
                            System.out.println("  " + i + ": "
                                + loadedFields.get(i).name + " (length=" + loadedFields.get(i).length + ")");
                        }
                    }
                    break;

                case "6":
                    System.out.println("Program exited.");
                    scanner.close();
                    return;

                default:
                    System.out.println("Wrong input.");
                    break;
            }
        }
    }
```

- 사용자 인터페이스: 메인 루프에서 사용자 입력에 따라 연결된 클래스의 기능을 사용한다.

2. createNewDataFileFromConfig(): 지정한 필드 정보 파일을 기반으로 파일 생성한다.

```
    private static boolean createNewDataFileFromConfig(String configPath, FileStructure fileStructure, MetadataManager metadataManager) {
        try (BufferedReader br = new BufferedReader(new FileReader(configPath))) {
            String line = br.readLine();
            if (line == null || line.trim().isEmpty()) {
                System.out.println("Config file empty or invalid.");
                return false;
            }
            line = line.trim();
            String[] parts = line.split(";");
            if (parts.length < 3) {
                System.out.println("Not enough fields in config. Format must be: fileName;fieldCount;fieldName...;fieldLength...");
                return false;
            }

            String fileName = parts[0].trim();
            int fieldCount = Integer.parseInt(parts[1].trim());

            int totalNeeded = 2 + fieldCount + fieldCount;  // "파일명,필드개수" + "필드Count" + "길이Count"
            if (parts.length != totalNeeded) {
                System.out.println("Mismatched field count vs actual input. Needed " + totalNeeded + " parts, got " + parts.length);
                return false;
            }

            List<String> fieldNames = new ArrayList<>();
            int idx = 2;
            for (int i = 0; i < fieldCount; i++) {
                fieldNames.add(parts[idx++].trim());
            }

            List<Integer> fieldLengths = new ArrayList<>();
            for (int i = 0; i < fieldCount; i++) {
                fieldLengths.add(Integer.parseInt(parts[idx++].trim()));
            }

            List<FileStructure.FieldInfo> fields = new ArrayList<>();
            for (int i = 0; i < fieldCount; i++) {
                fields.add(new FileStructure.FieldInfo(fieldNames.get(i), fieldLengths.get(i)));
            }

             String dataFile = fileName + ".dat";
             try (RandomAccessFile raf = new RandomAccessFile(dataFile, "rw")) {
                 fileStructure.writeHeader(raf, new FileStructure.Metadata(fields));
                 System.out.println("File '" + dataFile + "' created with user-defined fields from " + configPath);
                 metadataManager.loadMetadataFromFile(fileName);
             } catch (Exception e) {
                 System.out.println("Error creating file: " + e.getMessage());
                 return false;
             }

             return true;

        } catch (IOException e) {
            System.out.println("Could not read config file: " + e.getMessage());
            return false;
        } catch (NumberFormatException e) {
            System.out.println("Invalid integer value in config: " + e.getMessage());
            return false;
        }
    }
```

- 입력 파일 읽기: 사용자가 지정한 .txt 설정 파일에서 파일명, 필드 개수, 필드

이름, 필드 길이를 파싱한다.

- 유효성 검사: 필드 개수와 실제 필드 이름 및 길이 수가 일치하는지 검사하며, 불일치 시 생성을 중단한다.
- 메타데이터: 파싱한 정보를 기반으로 메타데이터를 구성하고 이를 헤더로 기록한 .dat 파일을 생성한다.
- 메모리에 로드: 생성된 파일의 메타데이터를 메모리에 적재한다.

4. 실행 결과

A. 테스트 데이터 변경 사항

i. 레코드 입력 테스트 데이터는 필드 B0(이전의 B)와 C0(이전의 C)의 크기가 4가 되었으므로 이에 맞게 변경하며, 헤더에 파일 명칭을 저장할 필요가 없으므로 삭제한다.

```
10,0001;X001;Y001|0002;null;Y002|0003;X003;null|0004;null;Y004|0005;X005;null|0006;X006;Y006|0007;null;null|0008;null;Y008|0009;X009;null|0010;X010;Y010
```

ii. 레코드 입력을 하기 전 데이터 파일(테이블)을 생성해야 할 필드 데이터가 필요하며, 본 테스트에서 사용할 필드 정보는 아래와 같다.

```
f1;3;A0;B0;C0;4;4;4
```

B. 기능별 실행 결과

i. 구동: 프로그램이 처음 실행되었을 때의 화면은 아래와 같다.

```
==== Database System ====
1. Create new data file (from config .txt)
2. Record Insertion (With data file)
3. Search Field
4. Search Record Range
5. Check Metadata(Memory)
6. Exit
Selection:
```

ii. 파일 생성: 사용자의 입력에 따른 파일을 생성한다.

```
==== Database System ====
1. Create new data file (from config .txt)
2. Record Insertion (With data file)
3. Search Field
4. Search Record Range
5. Check Metadata(Memory)
6. Exit
Selection: 1
Enter config text file path to create .dat: D:\SchoolHomework\4-1\Database\config.txt
File 'f1.dat' created with user-defined fields from D:\SchoolHomework\4-1\Database\config.txt
Metadata is loaded to memory from File f1.dat
Field numbers: 3
```

iii. 데이터 파일 이용한 레코드 삽입: 파일 명칭과 데이터 파일이 존재하는 경로를 입력하여 레코드를 삽입한다.

```
Selection: 2
Enter file name to insert into (without extension): f1
Record insertion data file path: D:\SchoolHomework\4-1\Database\testdata.txt
Records inserted into file 'f1.dat' successfully (no duplicate keys).
MySQL table 'f1' created/verified.
Insert SQL: INSERT INTO f1 (A0,B0,C0) VALUES(?,?,?)
All records inserted into MySQL table 'f1'.
Block:1 Offset:0 Search-key:0001
B0: X001
C0: Y001
Block:1 Offset:21 Search-key:0002
B0: null
C0: Y002
Block:1 Offset:42 Search-key:0003
B0: X003
C0: null
Block:1 Offset:63 Search-key:0004
B0: null
C0: Y004
Block:1 Offset:84 Search-key:0005
B0: X005
C0: null
Block:1 Offset:105 Search-key:0006
B0: X006
C0: Y006
Block:2 Offset:0 Search-key:0007
B0: null
C0: null
Block:2 Offset:21 Search-key:0008
B0: null
C0: Y008
Block:2 Offset:42 Search-key:0009
B0: X009
C0: null
Block:2 Offset:63 Search-key:0010
B0: X010
C0: Y010
```

iv. 필드 검색: 입력 받은 필드 명에 따른 각 필드의 레코드 값들을 출력한다.

```
Selection: 3
File name: f1
Search field name: A0
Block:1 Offset:0 Field A0 Value: 0001
Block:1 Offset:21 Field A0 Value: 0002
Block:1 Offset:42 Field A0 Value: 0003
Block:1 Offset:63 Field A0 Value: 0004
Block:1 Offset:84 Field A0 Value: 0005
Block:1 Offset:105 Field A0 Value: 0006
Block:2 Offset:0 Field A0 Value: 0007
Block:2 Offset:21 Field A0 Value: 0008
Block:2 Offset:42 Field A0 Value: 0009
Block:2 Offset:63 Field A0 Value: 0010
```

```
Selection: 3
File name: f1
Search field name: B0
Block:1 Offset:0 Field B0 Value: X001
Block:1 Offset:21 Field B0 Value: null
Block:1 Offset:42 Field B0 Value: X003
Block:1 Offset:63 Field B0 Value: null
Block:1 Offset:84 Field B0 Value: X005
Block:1 Offset:105 Field B0 Value: X006
Block:2 Offset:0 Field B0 Value: null
Block:2 Offset:21 Field B0 Value: null
Block:2 Offset:42 Field B0 Value: X009
Block:2 Offset:63 Field B0 Value: X010
```

```
Selection: 3
File name: f1
Search field name: C0
Block:1 Offset:0 Field C0 Value: Y001
Block:1 Offset:21 Field C0 Value: Y002
Block:1 Offset:42 Field C0 Value: null
Block:1 Offset:63 Field C0 Value: Y004
Block:1 Offset:84 Field C0 Value: null
Block:1 Offset:105 Field C0 Value: Y006
Block:2 Offset:0 Field C0 Value: null
Block:2 Offset:21 Field C0 Value: Y008
Block:2 Offset:42 Field C0 Value: null
Block:2 Offset:63 Field C0 Value: Y010
```

v. 레코드 범위 검색: 입력 받은 필드 값 범위 내의 레코드를 출력한다. 만약 레코드 범위에 아무런 데이터가 없을 경우 아무것도 출력하지 않는다.

```
Selection: 4
File name: f1
Search-key minimum value: 0002
Search-key maximum value: 0009
Block:1 Offset:21 Search-key:0002
B0: null
C0: Y002
Block:1 Offset:42 Search-key:0003
B0: X003
C0: null
Block:1 Offset:63 Search-key:0004
B0: null
C0: Y004
Block:1 Offset:84 Search-key:0005
B0: X005
C0: null
Block:1 Offset:105 Search-key:0006
B0: X006
C0: Y006
Block:2 Offset:0 Search-key:0007
B0: null
C0: null
Block:2 Offset:21 Search-key:0008
B0: null
C0: Y008
Block:2 Offset:42 Search-key:0009
B0: X009
C0: null
```

vi. 메타데이터의 저장 확인: 지목한 파일의 필드의 개수, 각 필드의 이름, 각 필드의 바이트 길이가 필드의 순서대로 출력된다.

```
Selection: 5
Enter file name (without extension) to check metadata: f1
Metadata is loaded to memory from File f1.dat
Field numbers: 3
List of fields in memory for file 'f1.dat':
  0: A0 (length=4)
  1: B0 (length=4)
  2: C0 (length=4)
```

5. 정확성 검증

A. 기능 검증

i. 위에서 테스트한 데이터 파일들의 총 구성은 다음과 같다.

f1		
A0	**B0**	**C0**
0001	X001	Y001
0002	null	Y002
0003	X003	null
0004	null	Y004
0005	X005	null
0006	X006	Y006
0007	null	null
0008	null	Y008
0009	X009	null
0010	X010	Y010

ii. 필드 파일 이용한 데이터 파일 생성

필드 파일로 데이터 파일을 생성했을 때의 결과는 다음과 같았다.

```
==== Database System ====
1. Create new data file (from config .txt)
2. Record Insertion (With data file)
3. Search Field
4. Search Record Range
5. Check Metadata(Memory)
6. Exit
Selection: 1
Enter config text file path to create .dat: D:\SchoolHomework\4-1\Database\config.txt
File 'f1.dat' created with user-defined fields from D:\SchoolHomework\4-1\Database\config.txt
Metadata is loaded to memory from File f1.dat
Field numbers: 3
```

만약 아래와 같이 필드 파일의 구성이 잘못되면 프로그램은 데이터 파일을 생성하지 않는다.

```
f1;3;A0;B0;C0;4;4;4;4
```

```
Selection: 1
Enter config text file path to create .dat: D:\SchoolHomework\4-1\Database\config.txt
Mismatched field count vs actual input. Needed 8 parts, got 9
Failed to create data file due to invalid config.
```

iii. 필드 정렬

레코드 파일의 구성을 아래와 같이 변경하고 테스트한다.

```
10,0001;X001;Y001|0002;null;Y002|0003;X003;null|0004;null;Y004|0005;X005;null|0009;X009;null|0010;X010;Y010|0006;X006;Y006|0007;null;null|0008;null;Y008
```

```
Selection: 2
Enter file name to insert into (without extension): f1
Record insertion data file path: D:\SchoolHomework\4-1\Database\testdata.txt
Records inserted into file 'f1.dat' successfully (no duplicate keys).
MySQL table 'f1' created/verified.
Insert SQL: INSERT INTO f1 (A0,B0,C0) VALUES(?,?,?)
All records inserted into MySQL table 'f1'.
Block:1 Offset:0 Search-key:0001
B0: X001
C0: Y001
Block:1 Offset:21 Search-key:0002
B0: null
C0: Y002
Block:1 Offset:42 Search-key:0003
B0: X003
C0: null
Block:1 Offset:63 Search-key:0004
B0: null
C0: Y004
Block:1 Offset:84 Search-key:0005
B0: X005
C0: null
Block:1 Offset:105 Search-key:0006
B0: X006
C0: Y006
Block:2 Offset:0 Search-key:0007
B0: null
C0: null
Block:2 Offset:21 Search-key:0008
B0: null
C0: Y008
Block:2 Offset:42 Search-key:0009
B0: X009
C0: null
Block:2 Offset:63 Search-key:0010
B0: X010
C0: Y010
```

해당 데이터에서 확인할 수 있듯이 출력된 데이터는 기존 레코드를 정렬하였음을 확인할 수 있다.

iv. 레코드 범위 검색

삽입된 레코드들에 대해 범위를 설정하고 검색했을 때의 결과는 다음과 같았다.

```
Selection: 4
File name: f1
Search-key minimum value: 0002
Search-key maximum value: 0009
Block:1 Offset:21 Search-key:0002
B0: null
C0: Y002
Block:1 Offset:42 Search-key:0003
B0: X003
C0: null
Block:1 Offset:63 Search-key:0004
B0: null
C0: Y004
Block:1 Offset:84 Search-key:0005
B0: X005
C0: null
Block:1 Offset:105 Search-key:0006
B0: X006
C0: Y006
Block:2 Offset:0 Search-key:0007
B0: null
C0: null
Block:2 Offset:21 Search-key:0008
B0: null
C0: Y008
Block:2 Offset:42 Search-key:0009
B0: X009
C0: null
```

해당 데이터에서 확인할 수 있듯이 출력된 데이터와 삽입된 데이터가 일치하는 것을 확인할 수 있다.

v. 메타데이터의 저장 확인

메타데이터를 출력했을 때의 결과 다음과 같았다.

```
Selection: 5
Enter file name (without extension) to check metadata: f1
Metadata is loaded to memory from File f1.dat
Field numbers: 3
List of fields in memory for file 'f1.dat':
  0: A0 (length=4)
  1: B0 (length=4)
  2: C0 (length=4)
```

해당 데이터에서 확인할 수 있듯이 출력된 데이터와 실제 메타데이터가 일치하는 것을 확인할 수 있다.

B. 구조 검증

i. null 검증

1. 해당 사항을 검증하기 위해 생성된 데이터의 .dat 파일을 Hex Editor로 확인하였다.
2. null이 없는 레코드의 데이터 형식은 다음과 같으며, 모든 값이 정상적으로 존재함을 확인할 수 있다.

```
00001070   7E 00 30 30 30 36 58 30 30 36 59 30 30 36 00 00   ~.0006X006Y006..
```

3. 반면 null이 존재하는 레코드의 데이터 형식은 다음과 같으며, null에 해당하는 부분이 정상적으로 비어 있음을 확인할 수 있다.

```
00001030   00 3F 04 30 30 30 33 58 30 30 33 00 00 00 00 00   .?.0003X003.....
00001040   00 00 01 00 00 00 54 02 30 30 30 34 59 30 30 34   ......T.0004Y004
```

4. 모든 레코드에 대해 null값 처리가 정상적으로 처리되었음을 확인할 수 있으므로 null은 DB 내에서 정상적으로 처리됨을 확인할 수 있다.

ii. 데이터 블록 검증

1. 해당 사항을 검증하기 위해 현재 BLOCK_SIZE를 4096에서 더 낮은 값으로 조정한다.
2. BLOCK_SIZE를 4096에서 128로 낮추고 데이터 파일 입력을 다시 수행할 시의 결과는 다음과 같다.

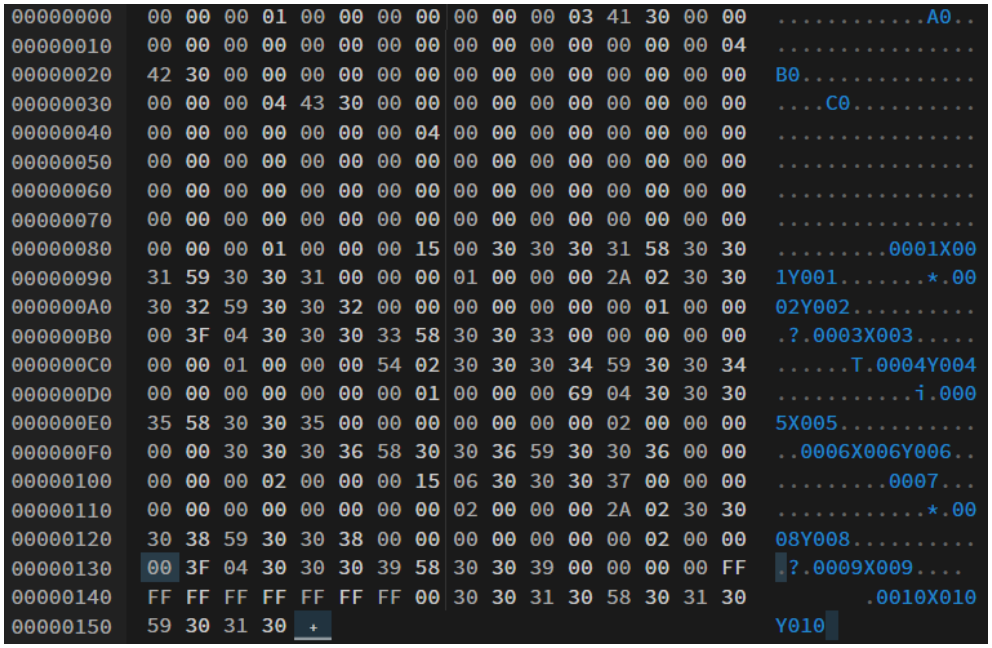
```
Block:1 Offset:0 Search-key:0001
B0: X001
C0: Y001
Block:1 Offset:21 Search-key:0002
B0: null
C0: Y002
Block:1 Offset:42 Search-key:0003
B0: X003
C0: null
Block:1 Offset:63 Search-key:0004
B0: null
C0: Y004
Block:1 Offset:84 Search-key:0005
B0: X005
C0: null
```

```
Block:1 Offset:105 Search-key:0006
B0: X006
C0: Y006
Block:2 Offset:0 Search-key:0007
B0: null
C0: null
Block:2 Offset:21 Search-key:0008
B0: null
C0: Y008
Block:2 Offset:42 Search-key:0009
B0: X009
C0: null
Block:2 Offset:63 Search-key:0010
B0: X010
C0: Y010
```

A. 해당 입력에서는 7~10번 레코드의 Block 값이 변한 것을 확인할 수 있다.

B. 이는 블록의 크기가 충분히 작기 때문에 해당 레코드들은 다음 블록에 저장된 것임을 알 수 있다.

3. 이를 Hex Editor로 확인한 세부 구성은 다음과 같다.



A. 0x00부터 0x7F까지의 Hex값을 확인해보면
   - 0x00, 0x00, 0x00, 0x01 → 첫 레코드 블록 = 1
   - 0x00, 0x00, 0x00, 0x00 → 첫 레코드 오프셋 = 0
   - 0x00, 0x00, 0x00, 0x03 → fieldCount = 3
   - A0, B0, C0에 대한 각각의 길이가 4, 4, 4으로 기록되어 있는 것을 확인할 수 있다.

B. 0x80부터 0xFF까지의 Hex값을 확인해보면
   - 1번 레코드의 경우 0x30, 0x30, 0x31의 값이 각각 0001, X001, Y001으로 삽입한 데이터와 일치하는 것을 확인할 수 있다.
   - 다른 데이터들 또한 동일한 값으로 저장된 것을 확인할 수 있다.
   - 따라서 필드값 들이 적절한 위치(오프셋)에 기록되어 있고 그 다음의 블록과 오프셋 또한 올바르게 설정되어 있음을 확인할 수 있다.

4. 따라서 데이터 블록은 정상적으로 처리되었다고 볼 수 있다.

iii. 메타데이터 검증

1. 메타데이터에 포함되어야 하는 정보는 필드의 개수, 각 필드의 이름, 각 필드의 바이트 길이, 레코드 포맷팅 시 필드의 순서다.
2. 메타데이터는 FileStructure의 Metadata 클래스에 저장되며, 이 객체는 헤더 블록으로서 .dat 파일 맨 앞에 저장되고 로딩된다.

```
public static class Metadata {
    int firstRecordBlock;
    int firstRecordOffset;
    int fieldCount;
    List<FieldInfo> fields;
}
```

3. 필드 정보에서 필드의 이름과 바이트 길이가 저장되며, 이 필드는 리스트이므로 인덱스 순서가 레코드 포맷의 필드 순서가 된다.

```
public static class FieldInfo {
    String name;
    int length;
}
```

4. 필드는 순서대로 순차적으로 기록되므로 순서가 보존되며, 필드 수, 이름, 길이 모두 기록된다.

```
buffer.putInt(meta.firstRecordBlock);
buffer.putInt(meta.firstRecordOffset);
buffer.putInt(meta.fieldCount);

for (FieldInfo field : meta.fields) {
    byte[] nameBytes =
field.name.getBytes(StandardCharsets.UTF_8);
    buffer.put(fixedName);
    buffer.putInt(field.length);
}
```

5. 앞서 언급한 설계대로라면 필드 A0, B0, C0가 각각 4, 4, 4바이트의 길이를 가지고 있으며, 이것이 정상적으로 출력되어야 한다.

```
Selection: 5
Enter file name (without extension) to check metadata: f1
Metadata is loaded to memory from File f1.dat
Field numbers: 3
List of fields in memory for file 'f1.dat':
  0: A0 (length=4)
  1: B0 (length=4)
  2: C0 (length=4)
```

A. 해당 이미지에서 모든 필드들이 순서(레코드 포맷)에 맞게 나열되어 있다.

B. 각 필드의 명칭(A0, B0, C0)가 각각의 필드에 대한 바이트 길이(4, 4, 4)과 함께 나열되어 있다.

C. 따라서 메타데이터는 정상적으로 처리되고 있다고 볼 수 있다.

6. 소스 코드

FileStructure.java

```
package main_package;

import java.io.RandomAccessFile;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.util.*;

public class FileStructure {
       private static final int BLOCK_SIZE = 4096;
       private static final int FIELD_NAME_SIZE = 16;

       // 필드 정보
       public static class FieldInfo {
           String name;
           int length;
           public FieldInfo(String name, int length) {
```

```
            this.name = name;
            this.length = length;
        }
    }

    // 메타데이터
    public static class Metadata {
        int firstRecordBlock;
        int firstRecordOffset;
        int fieldCount;
        List<FieldInfo> fields;

        public Metadata(List<FieldInfo> fields) {
            this.firstRecordBlock = -1;
            this.firstRecordOffset = -1;
            this.fieldCount = fields.size();
            this.fields = fields;
        }
    }

    // 레코드
    public static class Record {
        int nextRecordBlock;
        int nextRecordOffset;
        List<String> fieldValues;
        byte nullBitmap;

        public Record(List<String> fieldValues, byte nullBitmap) {
            this.nextRecordBlock = -1;
            this.nextRecordOffset = -1;
            this.fieldValues = fieldValues;
            this.nullBitmap = nullBitmap;
        }
    }

    // 메타데이터 헤더에 기록
    public void writeHeader(RandomAccessFile raf, Metadata meta)
    throws IOException {
        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
        buffer.putInt(meta.firstRecordBlock);
        buffer.putInt(meta.firstRecordOffset);
        buffer.putInt(meta.fieldCount);

        for (FieldInfo field : meta.fields) {
            byte[] nameBytes =
    field.name.getBytes(StandardCharsets.UTF_8);
            byte[] fixedName = new byte[FIELD_NAME_SIZE];
            System.arraycopy(nameBytes, 0, fixedName, 0,
    Math.min(nameBytes.length, FIELD_NAME_SIZE));
            buffer.put(fixedName);
            buffer.putInt(field.length);
        }

        raf.seek(0);
        raf.write(buffer.array());
    }
```

```
    // 헤더에서 메타데이터 읽기
    public Metadata readHeader(RandomAccessFile raf) throws
IOException {
        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
        raf.seek(0);
        raf.readFully(buffer.array());
        buffer.rewind();
        int firstRecordBlock = buffer.getInt();
        int firstRecordOffset = buffer.getInt();
        int fieldCount = buffer.getInt();
        List<FieldInfo> fields = new ArrayList<>();

        for (int i = 0; i < fieldCount; i++) {
            byte[] fixedName = new byte[FIELD_NAME_SIZE];
            buffer.get(fixedName);
            String name = new String(fixedName,
StandardCharsets.UTF_8).trim();
            int length = buffer.getInt();
            fields.add(new FieldInfo(name, length));
        }

        Metadata meta = new Metadata(fields);
        meta.firstRecordBlock = firstRecordBlock;
        meta.firstRecordOffset = firstRecordOffset;
        return meta;
    }

    // 레코드 크기 계산
    private int getRecordSize(Metadata meta) {
        int size = 4 + 4 + 1; // nextBlock + nextOffset + nullBitmap
        for (FieldInfo field : meta.fields) {
            size += field.length;
        }
        return size;
    }

    // 레코드 삽입
    public void insertRecord(String filename, Metadata meta, Record
newRecord) throws IOException {
        try (RandomAccessFile raf = new RandomAccessFile(filename,
"rw")) {
            int recordSize = getRecordSize(meta);
            int fileLength = (int) raf.length();
            int lastBlockStart = (fileLength / BLOCK_SIZE) *
BLOCK_SIZE;
            int offsetInLastBlock = fileLength % BLOCK_SIZE;

            int newBlock = fileLength / BLOCK_SIZE;
            int newOffset = offsetInLastBlock;

            // 새 블록으로 이동
            if (offsetInLastBlock + recordSize > BLOCK_SIZE) {
                newBlock++;
                newOffset = 0;
            }

            // 헤더에 위치 기록
```

```
            if (meta.firstRecordBlock == -1 && meta.firstRecordOffset
    == -1) {
                meta.firstRecordBlock = newBlock;
                meta.firstRecordOffset = newOffset;
                raf.seek(0);
                raf.writeInt(meta.firstRecordBlock);
                raf.writeInt(meta.firstRecordOffset);
            } else {
                  // 새로운 레코드로 연결
                int currBlock = meta.firstRecordBlock;
                int currOffset = meta.firstRecordOffset;

                while (true) {
                    raf.seek((long) currBlock * BLOCK_SIZE +
    currOffset);
                    int nextBlock = raf.readInt();
                    int nextOffset = raf.readInt();

                    if (nextBlock == -1 && nextOffset == -1) {
                        raf.seek((long) currBlock * BLOCK_SIZE +
    currOffset);
                        raf.writeInt(newBlock);
                        raf.writeInt(newOffset);
                        break;
                    } else {
                        currBlock = nextBlock;
                        currOffset = nextOffset;
                    }
                }
            }

            writeRecord(raf, newBlock, newOffset, meta, newRecord);
        }
    }

    // 레코드 위치 기록
    public void writeRecord(RandomAccessFile raf, int block, int
    offset, Metadata meta, Record record) throws IOException {
        int recordSize = getRecordSize(meta);
        raf.seek((long) block * BLOCK_SIZE + offset);
        ByteBuffer buffer = ByteBuffer.allocate(recordSize);

        buffer.putInt(record.nextRecordBlock);
        buffer.putInt(record.nextRecordOffset);
        buffer.put(record.nullBitmap);

        for (int i = 0; i < meta.fieldCount; i++) {
            if (((record.nullBitmap >> i) & 1) == 0) {
                String value = record.fieldValues.get(i);
                byte[] valueBytes =
    value.getBytes(StandardCharsets.UTF_8);
                byte[] fixedField = new
    byte[meta.fields.get(i).length];
                Arrays.fill(fixedField, (byte) ' ');
                System.arraycopy(valueBytes, 0, fixedField, 0,
    Math.min(valueBytes.length, fixedField.length));
                buffer.put(fixedField);
            }
```

```
        }

        raf.write(buffer.array());
    }
}
```

RecordBulkInserter.java

```
package main_package;

import java.io.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Types;
import java.util.*;

public class RecordBulkInserter {
    private FileStructure fileStructure = new FileStructure();
    private RecordSearcher recordSearcher = new RecordSearcher();
    private MetadataManager metadataManager = new MetadataManager();

    public void bulkInsertFromDataFile(String fileName, String
dataFilePath) {
        try (BufferedReader br = new BufferedReader(new
FileReader(dataFilePath))) {
            String dataFilename = fileName + ".dat";

            FileStructure.Metadata meta;
            try (RandomAccessFile raf = new
RandomAccessFile(dataFilename, "rw")) {
                meta = fileStructure.readHeader(raf);
            } catch (IOException e) {
                System.out.println("Failed to read header from file: "
+ fileName);
                return;
            }

            List<FileStructure.Record> recordList = new ArrayList<>();
            List<String> searchKeys = new ArrayList<>();

            String line;
            while ((line = br.readLine()) != null) {
                line = line.trim();
                if (line.isEmpty()) continue;

                String[] parts = line.split(",", 2);
                if (parts.length < 2) {
                    System.out.println("Invalid record line: " + line);
                    continue;
                }

                String recordsData = parts[1].trim();
                String[] recordStrArr = recordsData.split("\\|");
                for (String recStr : recordStrArr) {
                    String[] fieldValuesArr = recStr.split(";");
                    List<String> fieldValues = new ArrayList<>();
                    byte nullBitmap = 0;

                    for (int i = 0; i < fieldValuesArr.length; i++) {
```

```
                        String val = fieldValuesArr[i].trim();
                        if (val.isEmpty() ||
val.equalsIgnoreCase("null")) {
                            nullBitmap |= (1 << i);
                            fieldValues.add("");
                        } else {
                            fieldValues.add(val);
                        }
                    }

                    String searchKey = fieldValues.get(0);
                    if (!searchKey.isEmpty()) {
                        searchKeys.add(searchKey);
                    }

                    FileStructure.Record record = new
FileStructure.Record(fieldValues, nullBitmap);
                    recordList.add(record);
                }
            }

            // 탐색키 기준 정렬
            recordList.sort((r1, r2) ->
r1.fieldValues.get(0).compareTo(r2.fieldValues.get(0)));

            // 중복 키 검출
            for (int i = 0; i < recordList.size() - 1; i++) {
                String currKey = recordList.get(i).fieldValues.get(0);
                String nextKey = recordList.get(i +
1).fieldValues.get(0);
                if (currKey.equals(nextKey)) {
                    System.out.println("Insertion failed due to
duplicated keys in record file.");
                    return;
                }
            }

            for (FileStructure.Record rec : recordList) {
                fileStructure.insertRecord(dataFilename, meta, rec);
            }

            System.out.println("Records inserted into file '" +
dataFilename + "' successfully (no duplicate keys).");

            if (!searchKeys.isEmpty()) {
                Collections.sort(searchKeys);
                String minKey = searchKeys.get(0);
                String maxKey = searchKeys.get(searchKeys.size() - 1);
                System.out.println("Auto search: range " + minKey + " ~
" + maxKey);
                recordSearcher.searchRecordsByKeyRange(fileName,
minKey, maxKey);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
	/*
	public static void main(String[] args) {
	    String dataFilePath = "D:\SchoolHomework\4-
1\Database\testdata.txt";
	    RecordBulkInserter inserter = new RecordBulkInserter();
	    inserter.bulkInsertFromDataFile(dataFilePath);
	}
	*/
}
```

FieldSearcher.java

```
package main_package;

import java.io.*;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;

public class FieldSearcher {
	private static final int BLOCK_SIZE = 4096;
	private static final int FIELD_NAME_SIZE = 16;

	// 메타데이터 읽기
	public FileStructure.Metadata readHeader(RandomAccessFile raf)
throws IOException {
	    ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
	    raf.seek(0);
	    raf.readFully(buffer.array());
	    buffer.rewind();
	    int firstRecordBlock = buffer.getInt();
	    int firstRecordOffset = buffer.getInt();
	    int fieldCount = buffer.getInt();

	    FileStructure.Metadata meta = new FileStructure.Metadata(new
java.util.ArrayList<>());
	    meta.firstRecordBlock = firstRecordBlock;
	    meta.firstRecordOffset = firstRecordOffset;
	    meta.fieldCount = fieldCount;

	    // 필드 이름+길이 읽기
	    for (int i = 0; i < fieldCount; i++) {
	        byte[] fixedName = new byte[FIELD_NAME_SIZE];
	        buffer.get(fixedName);
	        String name = new String(fixedName,
StandardCharsets.UTF_8).trim();
	        int length = buffer.getInt();
	        meta.fields.add(new FileStructure.FieldInfo(name,
length));
	    }
	    return meta;
	}

	// 모든 레코드 값 검색
	public void searchField(String fileName, String searchField) {
	    String dataFilename = fileName + ".dat";
	    try (RandomAccessFile raf = new RandomAccessFile(dataFilename,
"r")) {
	        FileStructure.Metadata meta = readHeader(raf);
```

```
            // 인덱스 검색
            int fieldIndex = -1;
            for (int i = 0; i < meta.fieldCount; i++) {
                if
(meta.fields.get(i).name.equalsIgnoreCase(searchField)) {
                    fieldIndex = i;
                    break;
                }
            }
            if (fieldIndex == -1) {
                System.out.println("Cannot find field '" + searchField
+ "'.");
                return;
            }

            int block = meta.firstRecordBlock;
            int offset = meta.firstRecordOffset;

            // 레코드 순차 탐색
            while (block != -1 && offset != -1) {
                raf.seek((long) block * BLOCK_SIZE + offset);

                int nextRecordBlock = raf.readInt();
                int nextRecordOffset = raf.readInt();
                byte nullBitmap = raf.readByte();

                int dataOffset = 9; // pointer(8byte) +
nullBitmap(1byte)
                for (int i = 0; i < fieldIndex; i++) {
                    if (((nullBitmap >> i) & 1) == 0) {
                        dataOffset += meta.fields.get(i).length;
                    }
                }

                // null이 아닌 경우 해당 필드 읽기
                if (((nullBitmap >> fieldIndex) & 1) == 0) {
                    raf.seek((long) block * BLOCK_SIZE + offset +
dataOffset);
                    byte[] fieldBytes = new
byte[meta.fields.get(fieldIndex).length];
                    raf.read(fieldBytes);
                    String fieldValue = new String(fieldBytes,
StandardCharsets.UTF_8).trim();
                    System.out.println("Block:" + block + " Offset:" +
offset + " Field " + searchField + " Value: " + fieldValue);
                } else {
                    System.out.println("Block:" + block + " Offset:" +
offset + " Field " + searchField + " Value: null");
                }

                // 다음 레코드 이동
                block = nextRecordBlock;
                offset = nextRecordOffset;
            }

        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }

    /*
    public static void main(String[] args) {
        FieldSearcher searcher = new FieldSearcher();
        searcher.searchField("f1", "C");
    }
    */
}
```

RecordSearcher.java

```
package main_package;

import java.io.*;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
// import java.util.*;

public class RecordSearcher {
    private static final int BLOCK_SIZE = 4096;
    private static final int FIELD_NAME_SIZE = 16;

    // 헤더에서 메타데이터 읽기
    public FileStructure.Metadata readHeader(RandomAccessFile raf)
throws IOException {
        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
        raf.seek(0);
        raf.readFully(buffer.array());
        buffer.rewind();
        int firstRecordBlock = buffer.getInt();
        int firstRecordOffset = buffer.getInt();
        int fieldCount = buffer.getInt();

        FileStructure.Metadata meta = new FileStructure.Metadata(new
java.util.ArrayList<>());
        meta.firstRecordBlock = firstRecordBlock;
        meta.firstRecordOffset = firstRecordOffset;
        meta.fieldCount = fieldCount;

        for (int i = 0; i < fieldCount; i++) {
            byte[] fixedName = new byte[FIELD_NAME_SIZE];
            buffer.get(fixedName);
            String name = new String(fixedName,
StandardCharsets.UTF_8).trim();
            int length = buffer.getInt();
            meta.fields.add(new FileStructure.FieldInfo(name,
length));
        }
        return meta;
    }

    // 탐색키 범위 레코드 검색
    public void searchRecordsByKeyRange(String fileName, String
minKey, String maxKey) {
        String dataFilename = fileName + ".dat";
        try (RandomAccessFile raf = new RandomAccessFile(dataFilename,
"r")) {
```

```
            FileStructure.Metadata meta = readHeader(raf);
            int keyLength = meta.fields.get(0).length;

            int block = meta.firstRecordBlock;
            int offset = meta.firstRecordOffset;

            while (block != -1 && offset != -1) {
                raf.seek((long) block * BLOCK_SIZE + offset);
                int nextRecordBlock = raf.readInt();
                int nextRecordOffset = raf.readInt();
                byte nullBitmap = raf.readByte();

                byte[] keyBytes = new byte[keyLength];
                raf.read(keyBytes);
                String key = new String(keyBytes,
StandardCharsets.UTF_8).trim();
                // System.out.println("keylength = " + key.length());

                // 키가 범위 내에 있으면 출력
                if (key.compareTo(minKey) >= 0 && key.compareTo(maxKey)
<= 0) {
                    System.out.println("Block:" + block + " Offset:" +
offset + " Search-key:" + key);

                    int dataOffset = 9 + keyLength; // 포인터+비트맵+길이
                    for (int i = 1; i < meta.fieldCount; i++) { // 0번
키 제외
                        if (((nullBitmap >> i) & 1) == 0) {
                            raf.seek((long) block * BLOCK_SIZE + offset
+ dataOffset);
                            byte[] fieldBytes = new
byte[meta.fields.get(i).length];
                            raf.read(fieldBytes);
                            String fieldValue = new String(fieldBytes,
StandardCharsets.UTF_8).trim();
                            System.out.println(meta.fields.get(i).name +
": " + fieldValue);
                            dataOffset += meta.fields.get(i).length;
                        } else {
                            System.out.println(meta.fields.get(i).name +
": null");
                        }
                    }
                }

                block = nextRecordBlock;
                offset = nextRecordOffset;
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /*
    public static void main(String[] args) {
        RecordSearcher searcher = new RecordSearcher();
```

```
            searcher.searchRecordsByKeyRange("f1", "0001", "9999");
        }
        */
}
```

MetadataManger.java

```
package main_package;

import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class MetadataManager {

    private static final int BLOCK_SIZE = 4096;
    private static final int FIELD_NAME_SIZE = 16;

    // 필드 리스트
    private List<FileStructure.FieldInfo> fieldsInMemory = new
ArrayList<>();

    public void loadMetadataFromFile(String fileName) {
        String dataFile = fileName + ".dat";
        try (RandomAccessFile raf = new RandomAccessFile(dataFile, "r"))
{
            ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
            raf.seek(0);
            raf.readFully(buffer.array());
            buffer.rewind();

            int firstRecordBlock = buffer.getInt();          // 첫 레코드
블록 번호
            int firstRecordOffset = buffer.getInt();  // 첫 레코드 오프셋
            int fieldCount = buffer.getInt();                // 필드 개수

            List<FileStructure.FieldInfo> tempFields = new ArrayList<>();
            for (int i = 0; i < fieldCount; i++) {
                byte[] fixedName = new byte[FIELD_NAME_SIZE];
                buffer.get(fixedName);
                String name = new String(fixedName,
StandardCharsets.UTF_8).trim();
                int length = buffer.getInt();
                tempFields.add(new FileStructure.FieldInfo(name, length));
            }

            fieldsInMemory = tempFields; // 메모리 구조에 저장
            System.out.println("Metadata is loaded to memory from File "
+ fileName + ".dat");
            System.out.println("Field numbers: " + fieldCount);
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // 필드 정보 반환
    public List<FileStructure.FieldInfo> getFieldsInMemory() {
        return fieldsInMemory;
    }

    // 파일 헤더에 저장
    public void saveMetadataToFile(String fileName, int firstRecordBlock,
int firstRecordOffset) {
        String dataFile = fileName + ".dat";
        try (RandomAccessFile raf = new RandomAccessFile(dataFile, "rw"))
{
            ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);

            buffer.putInt(firstRecordBlock);
            buffer.putInt(firstRecordOffset);
            buffer.putInt(fieldsInMemory.size());

            for (FileStructure.FieldInfo field : fieldsInMemory) {
                byte[] nameBytes =
field.name.getBytes(StandardCharsets.UTF_8);
                byte[] fixedName = new byte[FIELD_NAME_SIZE];
                System.arraycopy(nameBytes, 0, fixedName, 0,
Math.min(nameBytes.length, FIELD_NAME_SIZE));
                buffer.put(fixedName);
                buffer.putInt(field.length);
            }

            raf.seek(0);
            raf.write(buffer.array());
            System.out.println("File " + fileName + ".dat's header
information has been updated.");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

MainApp.java

```
package main_package;

import java.util.*;
import java.io.*;

public class MainApp {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        FileStructure fileStructure = new FileStructure();
        RecordBulkInserter inserter = new RecordBulkInserter();
        FieldSearcher fieldSearcher = new FieldSearcher();
        RecordSearcher recordSearcher = new RecordSearcher();
        MetadataManager metadataManager = new MetadataManager();

        while (true) {
```

```
            System.out.println("\n==== Database System ====");
            System.out.println("1. Create new data file (from
config .txt)");
            System.out.println("2. Record Insertion (With data file)");
            System.out.println("3. Search Field");
            System.out.println("4. Search Record Range");
            System.out.println("5. Check Metadata(Memory)");
            System.out.println("6. Exit");
            System.out.print("Selection: ");
            String choice = scanner.nextLine();

            switch (choice) {
                case "1":
                    System.out.print("Enter config text file path to
create .dat: ");
                    String configPath = scanner.nextLine().trim();
                    boolean createSuccess =
createNewDataFileFromConfig(configPath, fileStructure, metadataManager);
                    if (!createSuccess) {
                        System.out.println("Failed to create data file due
to invalid config.");
                    }
                    break;

                case "2":
                    System.out.print("Enter file name to insert into
(without extension): ");
                    String insertFileName = scanner.nextLine().trim();
                    System.out.print("Record insertion data file path: ");
                    String dataFilePath = scanner.nextLine();
                    inserter.bulkInsertFromDataFile(insertFileName,
dataFilePath);
                    break;

                case "3":
                    System.out.print("File name: ");
                    String fNameField = scanner.nextLine();
                    System.out.print("Search field name: ");
                    String field = scanner.nextLine();
                    fieldSearcher.searchField(fNameField, field);
                    break;

                case "4":
                    System.out.print("File name: ");
                    String fNameRange = scanner.nextLine();
                    System.out.print("Search-key minimum value: ");
                    String minKey = scanner.nextLine();
                    System.out.print("Search-key maximum value: ");
                    String maxKey = scanner.nextLine();
                    recordSearcher.searchRecordsByKeyRange(fNameRange,
minKey, maxKey);
                    break;

                case "5":
                    System.out.print("Enter file name (without extension)
to check metadata: ");
                    String metaFileName = scanner.nextLine().trim();
                    metadataManager.loadMetadataFromFile(metaFileName);
```

```
                    List<FileStructure.FieldInfo> loadedFields =
metadataManager.getFieldsInMemory();
                    if (loadedFields.isEmpty()) {
                        System.out.println("No metadata loaded or file not
found.");
                    } else {
                        System.out.println("List of fields in memory for
file '" + metaFileName + ".dat':");
                        for (int i = 0; i < loadedFields.size(); i++) {
                            System.out.println("  " + i + ": "
                                + loadedFields.get(i).name + " (length=" +
loadedFields.get(i).length + ")");
                        }
                    }
                    break;

                case "6":
                    System.out.println("Program exited.");
                    scanner.close();
                    return;

                default:
                    System.out.println("Wrong input.");
                    break;
            }
        }
    }

    // 필드 정보 입력
    private static boolean createNewDataFileFromConfig(String configPath,
FileStructure fileStructure, MetadataManager metadataManager) {
        try (BufferedReader br = new BufferedReader(new
FileReader(configPath))) {
            String line = br.readLine();
            if (line == null || line.trim().isEmpty()) {
                System.out.println("Config file empty or invalid.");
                return false;
            }
            line = line.trim();
            String[] parts = line.split(";");
            if (parts.length < 3) {
                System.out.println("Not enough fields in config. Format
must be: fileName;fieldCount;fieldName...;fieldLength...");
                return false;
            }

            String fileName = parts[0].trim();
            int fieldCount = Integer.parseInt(parts[1].trim());

            int totalNeeded = 2 + fieldCount + fieldCount;  // "파일명,필
드개수" + "필드Count" + "길이Count"
            if (parts.length != totalNeeded) {
                System.out.println("Mismatched field count vs actual
input. Needed " + totalNeeded + " parts, got " + parts.length);
                return false;
            }

            List<String> fieldNames = new ArrayList<>();
```

```
            int idx = 2;
            for (int i = 0; i < fieldCount; i++) {
                fieldNames.add(parts[idx++].trim());
            }

            List<Integer> fieldLengths = new ArrayList<>();
            for (int i = 0; i < fieldCount; i++) {
                fieldLengths.add(Integer.parseInt(parts[idx++].trim()));
            }

            List<FileStructure.FieldInfo> fields = new ArrayList<>();
            for (int i = 0; i < fieldCount; i++) {
                fields.add(new FileStructure.FieldInfo(fieldNames.get(i),
fieldLengths.get(i)));
            }

            String dataFile = fileName + ".dat";
            try (RandomAccessFile raf = new RandomAccessFile(dataFile,
"rw")) {
                fileStructure.writeHeader(raf, new
FileStructure.Metadata(fields));
                System.out.println("File '" + dataFile + "' created with
user-defined fields from " + configPath);
                metadataManager.loadMetadataFromFile(fileName);
            } catch (Exception e) {
                System.out.println("Error creating file: " +
e.getMessage());
                return false;
            }

            return true;

        } catch (IOException e) {
            System.out.println("Could not read config file: " +
e.getMessage());
            return false;
        } catch (NumberFormatException e) {
            System.out.println("Invalid integer value in config: " +
e.getMessage());
            return false;
        }
    }
}
```