〈레포트〉

설계 과제
Part 2

강의명: 데이타베이스시스템
담당교수: 강현철 교수님
작성자: 신성섭
학번: 20226XX2
학과: 소프트웨어학부

1. 데이터 및 테스트 시나리오

-Join 테스트용 데이터

* book

Book_ID (5)	Title(15)	Author_ID(5)	price(5)
10001	Secret Garden	99999	30000
10002	Lost Horizon	99998	20000
10003	Starry Night	99997	15000
10004	New Year	99996	42000
10005	Happy Bear	99995	23000
10006	Scarry World	99994	12000
10007	Rain Rain Rain	99993	16000
10008	Big Boom	99992	24000
10009	Boo king	99991	37000
10010	New town	99990	14000

* Inventory

Inventory_ID (5)	Branch_ID(5)	BOOK_ID(5)	quantity(5)
50001	30003	10001	100
50002	30001	10002	150
50003	30005	10001	1000
50004	30002	10004	50
50005	30007	10005	1500
50006	30001	10006	1000
50007	30005	10010	600
50008	30007	10008	300
50009	30009	10003	300
50010	30002	10009	1200

- 테스트 시나리오

* part1과 마찬가지로 block의 size가 100bytes 라고 가정한다.
* 각 partition의 개수는 4개로 가정한다.
* hash index를 생성하기 위한 hash_bucket의 개수는 3으로 가정한다.

1) 위의 두 테이블 book과 inventory를 대상으로 join 연산을 수행

이때 book을 저장하는 파일의 중간에는 2개의 L$pos(삭제된 레코드가 있었던 공간)가 있는 상태로 테스트하고, inventory의 경우 순차적으로 가득 차 있는 파일인 상태로 테스트를 수행한다.

다음 sql 구문으로 나오는 테스트의 결과를 확인한다.

```
select *
from book
inner join inventory
on book.book_id = inventory.book_id
```

이번에는 두 테이블의 순서를 바꿔 다음 명령어의 테스트를 확인한다.

```
select *
from inventory
inner join book
on inventory.book_id = book.book_id
```

테스트 진행이 완료 후 임시파일을 확인해 본다.
레코드의 길이가 다르고 파일의 상태가 다른 두 테이블을 조인한 결과 테스트를 통해 구현한 프로그램이 일반성을 가짐을 확인 할 수 있다.

2. 설계

- 프로그램 기능 메뉴창

```
**** 기능 목록****
1. 관계 테이블 생성(create table)
2. 튜플 삽입(insert record)
3. 튜플 삭제(delete tuple)
4. 단일 테이블 튜플 검색(select tuple - one Table)
5. 조인 연산 수행(Hash Join - Two table)
6. 프로그램 종료(exit program)
```

part1과 비교 했을 때, 5번 조인 연산 수행이 추가되었다. 이전에 수행하던 검색 기능은 명확성을 위해 '단일 테이블 튜플 검색'으로 기능명을 바꾼다.

- part2에서 추가적으로 구현해야 하는 기능

1) join sql 사용자 입력 기능 및 join sql 파싱
2) 2가지의 서로 다른 종류의 hashFunction 구현
3) hash용 Bucket 클래스 구현
4) Partition 수행 메소드
5) HashIndex 구현 메소드
6) Join 수행 메소드

- 블록단위 I/O 구현을 위해 필요한 메소드

part1 에서 구현한 readBlock(), writeBlock() 메소드를 그대로 재사용하여 블록단위 I/O을 수행한다.

- 해쉬 조인을 위해 필요한 Hash 함수

1) partition 형성 용

입력받은 문자열의 각 문자의 아스키코드 값을 모두 더한 후 특정 숫자로 모듈러 연산을 수행한 결과를 해쉬 값으로 이용한다.

2) Hash Index용

자바 자체에서 제공하는 hashCode()를 이용하여 값을 얻은 후 특정 숫자로 모듈러 연산을 수행한 결과를 해쉬 값으로 이용한다.

- 관계 DB 시스템의 기능별 사용자 인터페이스

* 파란색 글자가 콘솔에 나오면 유저가 붉은 글자 형식으로 입력
* 1~4번 기능은 part1에서 구현한 인터페이스와 동일하다.

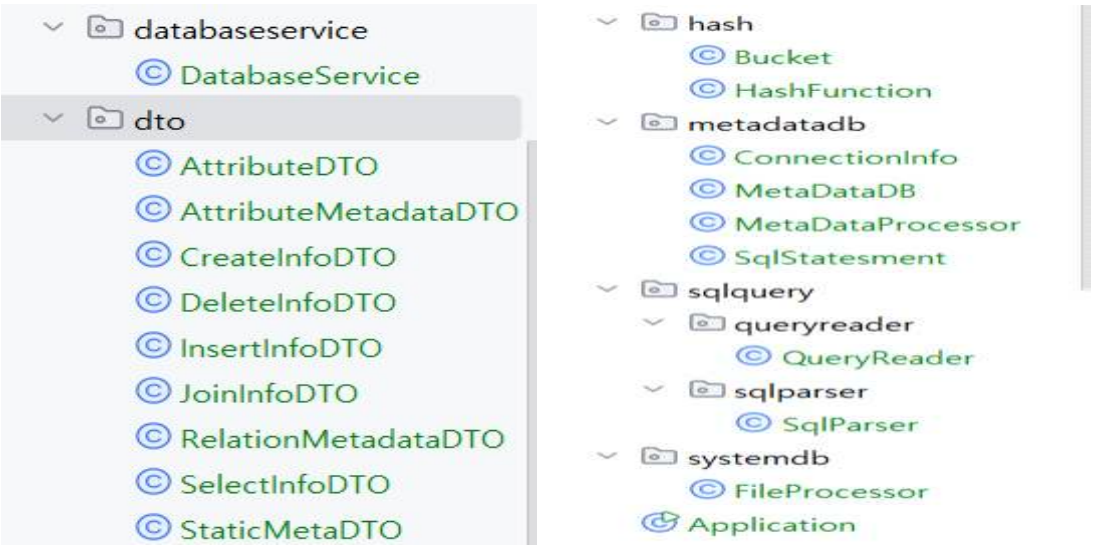
1) hash join 연산 수행

메뉴창에서 5 (조인 연산 수행(Hash join - two table))을 입력한다.
select * form 이 나오면 유저가 조인하기를 원하는 테이블 이름을 입력하고 엔터를 친다. 그 후 inner join 이 나오면 조인하기를 원하는 다른 테이블을 입력하고 엔터를 친다. 마지막으로 on 이 나오면 join 조건을 A.column = B. column 형식으로 입력 해준다.

사용 예시)
select *
from book (엔터 입력)
inner join inventory (엔터 입력)
on book.book_id = inventory.book_id (엔터 입력)

3. 구현

1) 코드 파일 디렉터리 구조



part1에서 구현한 디렉터리 구조와 나머지 부분은 동일한데,
Hash와 관련된 것들을 관리하기 쉽게 하기 위해서 hash package를 추가해 주었다.

2) part2에서 추가된 주요 구현 로직.

(1) FileProcessor

```
switch(select_num) {
    case 1: // 튜플 생성
        databaseService.createTable();
        break;
    case 2: // 튜플 삽입
        databaseService.insertTuple();
        break;
    case 3: // 튜플 삭제
        databaseService.deleteTuple();
        break;
    case 4: // 튜플 검색
        databaseService.selectTuple();
        break;
    case 5: // Hash Join 수행
        databaseService.joinTuple();
        break;
    case 6:
```

Hash Join을 수행하기 위한 기능 선택인 '5'가 추가되었다.

(2) DatabaseService

```
public void joinTuple(){  1 usage  new *
    JoinInfoDTO joinInfoDTO = sqlReader.joinSQL();
    if(joinInfoDTO == null){
        System.out.println("SQL 구문을 잘못 입력하였습니다. ");
        System.out.println();
        return;
    }

    Object[] oneMetadatas = getMetaDatas(joinInfoDTO.getTableName());
    Object[] anotherMetadatas = getMetaDatas(joinInfoDTO.getAnotherTableName());

    if(oneMetadatas != null && anotherMetadatas != null){
        fileProcessor.joinTuple(joinInfoDTO,(RelationMetadataDTO)oneMetadatas[0], (List<AttributeMetadataDTO>)oneMetadatas[1],(StaticMetaDTO)oneMetadatas[2],
                (RelationMetadataDTO)anotherMetadatas[0], (List<AttributeMetadataDTO>)anotherMetadatas[1],(StaticMetaDTO)anotherMetadatas[2]);

    }
}
```

유저로 부터 입력받은 테이블 정보를 이용해 관련된 메타데이터들을 얻은 후, fileProcessor.joinTuple()을 호출한다.

(3) QueryReader + SqlParser

-QueryReader의 joinSQL()

```
//Join SQL 문을 입력 받고 파싱 후 JoinInfoDTO를 반환하는 함수
public JoinInfoDTO joinSQL(){  1 usage  new *
    System.out.println("select * ");
    System.out.print("from ");
    String tableName = scanner.nextLine().trim();

    System.out.print("inner join ");
    String AnotherTableName = scanner.nextLine().trim();

    System.out.print("on ");
    String joinCondition = scanner.nextLine().trim();
    System.out.println();

    return SqlParser.parsingJoinSQL(tableName,AnotherTableName,joinCondition);
}
```

SQL 형식의 구문을 입력받고 입력받은 정보를 파싱 함수를 이용해 파싱한 후 JoinInfoDTO를 반환한다.

-SqlParser의 parsingJoinSQL

```
public static JoinInfoDTO parsingJoinSQL(String tableName, String anotherTableName, String joinCondition) {  1 usage  new *
    // = 기준으로 조인 조건을 분리
    String[] conditions = joinCondition.split( regex: "=");

    // 공백 제거
    String leftPart = conditions[0].trim();
    String rightPart = conditions[1].trim();

    // . 을 기준으로 Table과 Column 분리
    String[] leftParts = leftPart.split( regex: "\\.");
    String leftTable = leftParts[0].trim();
    String leftColumn = leftParts[1].trim();

    String[] rightParts = rightPart.split( regex: "\\.");
    String rightTable = rightParts[0].trim();
    String rightColumn = rightParts[1].trim();

    //System.out.println(tableName + anotherTableName + leftTable + leftColumn + rightTable + rightColumn); //디버깅 확인용

    // 정보를 DTO에 매핑
    if (tableName.equals(leftTable) && anotherTableName.equals(rightTable)) {
        return new JoinInfoDTO(tableName, anotherTableName, leftColumn, rightColumn);
    }else if (tableName.equals(rightTable) && anotherTableName.equals(leftTable)) {
        return new JoinInfoDTO(tableName, anotherTableName, rightColumn, leftColumn);
    }else{ // 잘못된 Sql 구문 입력 (error)
        return null;
    }

}
```

입력받은 데이터들을 파싱해 테이블 이름과 조인 컬럼을 각각 얻어내고 이 정보들을 JoinInfoDTO에 담아 반환 한다.

(4) HashFunction

```
public class HashFunction {  4 usages  new *

    // partition을 만들 때 사용하는 해시 함수
    public static int partitionHash(String input, int range) {  1 usage  new *
        int hashValue = 0;
        for (char charValue : input.toCharArray()) {
            hashValue += charValue;
        }
        return Math.abs(hashValue % range);
    }

    // 해시 인덱스를 구축하는 데 사용할 해시 함수
    public static int indexHash(String input, int range) {  2 usages  new *
        int hashValue = input.hashCode();
        return Math.abs(hashValue % range);
    }

}
```

'2. 설계'에서 계획한대로 Partition용 해시 함수와 hashIndex용 해쉬함수를 각각 구현해 주었다.

(5) Bucket

```
public class Bucket {  8 usages  new *
    private final String key;  2 usages
    private final int blockNum;  2 usages
    private final int offset;  2 usages
    private final int recordLength;  2 usages

    public Bucket(String key, int blockNum, int offset, int recordLength) {
        this.key = key;
        this.blockNum = blockNum;
        this.offset = offset;
        this.recordLength = recordLength;
    }
```

해쉬 인덱스의 각각의 버킷의 엔트리에 해당하며, key 값과, 메모리 버퍼에 저장된 레코드의 위치를 가리키기 위한 blockNum 과 offset, 레코드의 길이를 저장하는 recordLength로 필드가 이루어 져 있다.

(6) FileProcessor

- joinTuple() 메소드

```java
public void joinTuple(JoinInfoDTO joinInfoDTO, RelationMetadataDTO onerelationMetadataDTO, List<AttributeMetadataDTO> oneAttributeMetadatas, StaticMetaDTO oneStatic
    // Partition 수행
    List<String> oneTempFiles = makePartitionFiles(joinInfoDTO.getTableJoinColumn(), onerelationMetadataDTO, oneAttributeMetadatas, oneStaticMetaDTO);
    List<String> anotherTempFiles = makePartitionFiles(joinInfoDTO.getAnotherTableJoinColumn(), anotherrelationMetadataDTO, anotherAttributeMetadatas, anotherStatic

    // 모든 컬럼명 출력
    printAllColumnName(oneAttributeMetadatas, anotherAttributeMetadatas);

    // 각 partition 마다 join 수행
    for (int i = 0; i < PARTIAL_COUNT; i++) {
        // Hash Index 초기화
        List<List<Bucket>> bucketIndex = initializeBucketIndex(HASH_BUCKET_COUNT);

        // Ti 임시파일을 읽고 데이터를 메모리로 로드하기
        String anotherFilePath = anotherTempFiles.get(i);
        List<String> memoryBuffer = loadFileToMemory(anotherFilePath);

        // 인덱스 구축을 위해 필요한 메타데이터 검색
        int anotherRecordSize = anotherStaticMetaDTO.getRecordSize();
        int[] anotherColumnInfo = getColumnPositionAndSize(anotherAttributeMetadatas, joinInfoDTO.getAnotherTableJoinColumn());

        // 출력을 위한 메타데이터 검색
        List<Integer> oneColumnPositions = getColumnPositions(oneAttributeMetadatas);
        List<Integer> anotherColumnPositions = getColumnPositions(anotherAttributeMetadatas);

        // HashIndex 구축
        buildHashIndex(bucketIndex, memoryBuffer, anotherColumnInfo, anotherRecordSize);

        // 조인 수행
        performJoin(oneTempFiles.get(i), bucketIndex, memoryBuffer, oneStaticMetaDTO,
                oneAttributeMetadatas, joinInfoDTO.getTableJoinColumn(), oneColumnPositions, anotherColumnPositions);
```

HashJoin을 수행하는 메소드이다. 각각의 파일을 Partition을 수행한 후, partition 과정에서 생성된 임시 파일명들을 받아온다. 각 Partition 끼리 Join 연산을 수행하는데, 우선 구축입력을 이용해 해쉬 인덱스를 구축한다. 그 후 탐색입력의 각 튜플의 순차적으로 해쉬를 수행하면서 해쉬인덱스를 이용해 매치되는 튜플을 찾고 join 결과를 출력한다.

- makePartitionFiles() 메소드

```java
public List<String> makePartitionFiles(String partialColumn, RelationMetadataDTO relationMetadataDTO,
    String originFilePath = relationMetadataDTO.getFileLocation();
    long originEOFPosition = getEndOfFilePosition(originFilePath);
    int originBlockNum = 0;

    //필요한 메타데이터 정보 초기화
    int recordSize = staticMetaDTO.getRecordSize();
    int[] columnInfo = getColumnPositionAndSize(attributeMetadatas,partialColumn);
    int columnPosition = columnInfo[0];
    int columnSize = columnInfo[1];

    // 임시 파일에 쓰기위한 변수들 초기화
    List<String> temporaryFiles = initializeTemporaryFiles(originFilePath);
    List<String> tempMemoryBuffers = initializeTempMemoryBuffers();
    List<Integer> tempOffsets = initializeTempOffsets();
    List<Integer> tempBlockNums = initializeTempBlockNums();

    //Partition 수행
    while ((long) originBlockNum * BLOCK_SIZE < originEOFPosition) {
        String readingData = readBlock(originFilePath,  seekPosition: (long) originBlockNum * BLOCK_SIZE);
        partitionBlock(recordSize, columnPosition, columnSize, temporaryFiles,
                       tempMemoryBuffers, tempOffsets, tempBlockNums, readingData);
        originBlockNum++;
    }

    //버퍼에 남아있는 데이터를 Disk에 쓰기
    writeRemainingData(temporaryFiles, tempMemoryBuffers, tempOffsets, tempBlockNums);

    // 임시 파일명들 반환
    return temporaryFiles;
```

블록단위로 파일에서 정보를 읽어온 후 읽어온 Block에 들어있는 record에 대해 Partition을 수행한다. 파일을 모두 다 읽은 후에 버퍼에 남아있는 데이터들은 추가적으로 임시파일에 쓴 후 임시 파일명들을 반환한다.

-partitionBlock() 메소드

```
//한블록을 partition 수행하는 함수
private void partitionBlock(int recordSize, int columnPosition, int columnSize, List<String> temporaryFile,  1 usage  new *
                            List<String> tempMemoryBuffers, List<Integer> tempOffsets, List<Integer> tempBlockNums, String read
    int recordOffset = 0;
    while (recordOffset <= BLOCK_SIZE - recordSize) {
        String record = readingData.substring(recordOffset, recordOffset + recordSize);
        if (!record.contains("L$pos") && !record.trim().isEmpty()) {
            String columnValue = record.substring(columnPosition, columnPosition + columnSize);
            int hashValue = HashFunction.partitionHash(columnValue, PARTIAL_COUNT); // ColumnValue를 해쉬함수의 input으로 넣어 part
            // System.out.println(columnValue+hashValue);  //디버깅 확인용
            saveRecordInPartition(recordSize,temporaryFile, tempMemoryBuffers, tempOffsets, tempBlockNums, hashValue, record);
        }
        recordOffset += recordSize;
    }
}
```

한 블록 안에 들어있는 레코드들을 각각 해시함수를 수행함으로 partition을 수행한다. 해시값에 해당하는 memoryBuffer에 저장하며, memoryBuffer가 가득 차면 임시파일에 메모리 한 블록 전체를 쓴다.(블록단위 writing)

-loadFileToMemory() 메소드

```
//파일의 내용 전체를 메모리로 로드 하는 함수
private List<String> loadFileToMemory(String filePath) {  1 usage  new *
    List<String> memoryBuffer = new ArrayList<>();
    long EOFPosition = getEndOfFilePosition(filePath);
    int blockNum = 0;

    while ((long) blockNum * BLOCK_SIZE < EOFPosition) {
        memoryBuffer.add(readBlock(filePath,  seekPosition: (long) blockNum * BLOCK_SIZE));
        blockNum++;
    }
    return memoryBuffer;
}
```

파일 전체를 메모리로 로드한다. 이때 블록단위로 읽어서 memoryBuffer에 저장한다.

- buildHashIndex() 메소드

```
//해쉬 인덱스 구축
private void buildHashIndex(List<List<Bucket>> bucketIndex, List<String> memoryBuffer, int[] columnInfo, int recordSize) {  1 usage  new *
    int columnPosition = columnInfo[0];
    int columnSize = columnInfo[1];

    for (int blockNum = 0; blockNum < memoryBuffer.size() ; blockNum++) {
        String readingData = memoryBuffer.get(blockNum);
        int recordOffset = 0;
        while (recordOffset <= BLOCK_SIZE - recordSize) {
            String record = readingData.substring(recordOffset, recordOffset + recordSize);
            if (!record.trim().isEmpty()) {
                String columnValue = record.substring(columnPosition, columnPosition + columnSize);
                int hashValue = HashFunction.indexHash(columnValue, HASH_BUCKET_COUNT); // ColumnValue를 해시 함수의 입력으로 넣어 해시 값 얻기
                bucketIndex.get(hashValue).add(new Bucket(columnValue, blockNum, recordOffset, recordSize));
            }
            recordOffset += recordSize;
        }
    }
}
```

memoryBuffer에 들어있는 record를 모두읽어서 해시를 수행하여 해시인덱스를 구축한다.

-performJoin 메소드

```
private void performJoin(String filePath, List<List<Bucket>> bucketIndex, List<String> memoryBuffer, StaticMetaDTO staticMetaDTO,  1 usage  new *
    long EOFPosition = getEndOfFilePosition(filePath);
    int recordSize = staticMetaDTO.getRecordSize();
    int[] columnInfo = getColumnPositionAndSize(attributeMetadatas, joinColumn);

    int columnPosition = columnInfo[0];
    int columnSize = columnInfo[1];

    int blockNum = 0;
    while ((long) blockNum * BLOCK_SIZE < EOFPosition) {
        String readingData = readBlock(filePath,  seekPosition: (long) blockNum * BLOCK_SIZE);
        int recordOffset = 0;
        while (recordOffset <= BLOCK_SIZE - recordSize) {
            String record = readingData.substring(recordOffset, recordOffset + recordSize);
            if (!record.trim().isEmpty()) {
                String columnValue = record.substring(columnPosition, columnPosition + columnSize);
                int hashValue = HashFunction.indexHash(columnValue, HASH_BUCKET_COUNT); // ColumnValue를 해시 함수의 입력으로 넣어 해시 값 얻기

                for (Bucket bucket : bucketIndex.get(hashValue)) {
                    if (bucket.getKey().equals(columnValue)) {
                        printRecord(record, oneColumnPositions);
                        String anotherRecord = memoryBuffer.get(bucket.getBlockNum()).substring(bucket.getOffset(), bucket.getOffset() + bucket.getRecordLength());
                        printRecord(anotherRecord, anotherColumnPositions);
                        System.out.println();
                    }
                }
            }
            recordOffset += recordSize;
        }
        blockNum++;
    }
```

탐색 입력 릴레이션을 블록단위로 읽어오면서, 한 블록 안에 있는 각각의 레코드들에 대해 해시를 수행한다. 해시 수행 결과 나오는 해쉬값에 해당하는 Bucket을 순차적으로 탐색하면서 동일한 Key 값이 있는지 확인하고 동일한 Key 값을 가진다면 레코드를 출력한다.

- PartialCount & BucketCount

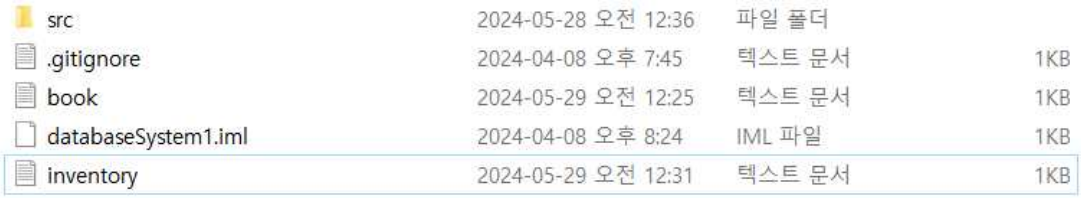
```
public class FileProcessor {  3 usages  new *

    private final int BLOCK_SIZE = 100; // 블록 input/output 단위 크기  30 usages
    private final int PARTIAL_COUNT = 4; // Join 연산 수행시 릴레이션의 분할 개수  7 usages
    private final int HASH_BUCKET_COUNT = 3; //hash bucket의 개수  3 usages
```

Join 연산 수행 시 릴레이션의 분할 개수 및 hash Bucket의 개수를 상수로 관리하여 다른 값으로 변경이 쉽도록 해주었다.

4. 테스트 및 정확성 검증

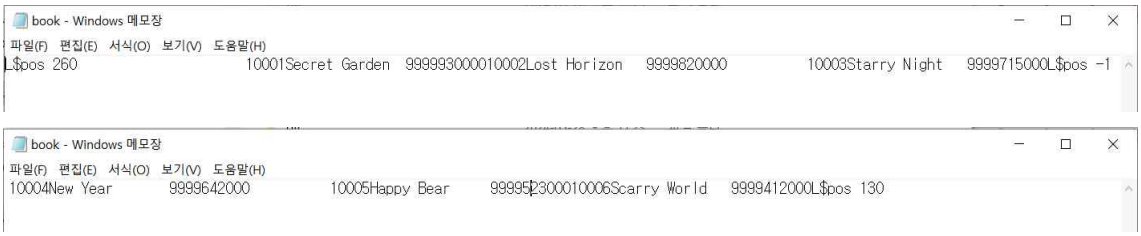
1) Test 초기 설정

- book.txt 와 inventory.txt 파일



book.txt 파일과 inventory.txt 파일이 저장되어 있는 상태이다.

-book.txt 파일



‘ 1. join용 Test 데이터’ 대로 book.txt 파일에 정보가 저장되어 있는 상태이고 2개의 L$pos가 포함되어 있는 상태이다.

- inventory.txt 파일

‘ 1. join용 Test 데이터’ 대로 inventory.txt 파일에 정보가 저장되어 있는 상태이고 순차적으로 레코드들이 저장되어 있는 상태이다.

- book table 데이터 값

```
기능 선택:4

select *
from book
where True

book_id             title               author_id           price
10001               Secret Garden       99999               30000
10002               Lost Horizon        99998               20000
10003               Starry Night        99997               15000
10004               New Year            99996               42000
10005               Happy Bear          99995               23000
10006               Scarry World        99994               12000
10007               Rain Rain Rain      99993               16000
10008               Big Boom            99992               24000
10009               Boo King            99991               37000
10010               New town            99990               14000
```

1. join용 Test 데이터’ 와 동일하다.

- inventory table 데이터 값

```
기능 선택:4

select *
from inventory
where True
```

inventory_id	branch_id	book_id	quantity
50001	30003	10001	100
50002	30001	10002	150
50003	30005	10001	1000
50004	30002	10004	50
50005	30007	10005	1500
50006	30001	10006	1000
50007	30005	10010	600
50008	30007	10008	300
50009	30009	10003	300
50010	30002	10009	1200

1. join용 Test 데이터’ 와 동일하다.

2) Test 1

```
기능 선택:5

select *
from book
inner join inventory
on book.book_id = inventory.book_id
```

book_id	title	author_id	price	inventory_id	branch_id	book_id	quantity
10003	Starry Night	99997	15000	50009	30009	10003	300
10004	New Year	99996	42000	50004	30002	10004	50
10008	Big Boom	99992	24000	50008	30007	10008	300
10001	Secret Garden	99999	30000	50001	30003	10001	100
10001	Secret Garden	99999	30000	50003	30005	10001	1000
10005	Happy Bear	99995	23000	50005	30007	10005	1500
10009	Boo King	99991	37000	50010	30002	10009	1200
10010	New town	99990	14000	50007	30005	10010	600
10002	Lost Horizon	99998	20000	50002	30001	10002	150
10006	Scarry World	99994	12000	50006	30001	10006	1000

inventory Table을 구축입력으로 했을 때 조인결과이다. 두 book_id의 컬럼 값이 동일 한 것을 통해 inventory의 모든 레코드들이 올바르게 book Table과 조인되었음을 확인 할 수 있다. inventory에 book_id가 10001 인 레코드는 2개 있는 데, 이 레코드들 모두 (10001, Secret Garden, 99999, 30000)과 조인 되었으므로 같은 컬럼값을 가지는 것이 복수개 있는 상황에서도 올바르게 동작함을 알 수 있다.

inventory의 book_id는 외래키고 book의 book_id는 주키이므로 외래키 제약 조건에 따라 inventory의 book_id 값은 반드시 book의 book_id에 나타나야 한다. 따라서 두 테이블을 조인한다면 inventory의 모든 record는 반드시 join 결과에 나타나야 한다. 위의 결과는 이를 만족하므로(inventory의 레코드 개수인 10개가 모두 나타나므로) 해시 조인이 올바르게 동작하였음을 알 수 있다.

조인이 올바르게 수행되었으므로, 원본 파일에 L$pos가 중간에 포함되어 있는지에 상관없이 올바르게 동작함을 확인 할 수 있다.

3) Test 2

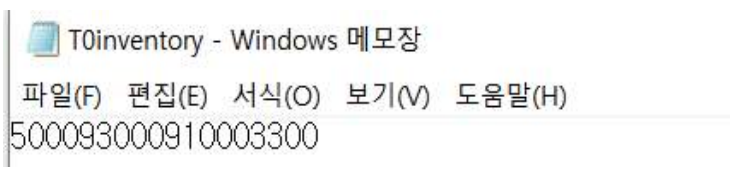
```
기능 선택:5

select *
from inventory
inner join book
on inventory.book_id = book.book_id
```

inventory_id	branch_id	book_id	quantity	book_id	title	author_id	price
50009	30009	10003	300	10003	Starry Night	99997	15000
50004	30002	10004	50	10004	New Year	99996	42000
50008	30007	10008	300	10008	Big Boom	99992	24000
50001	30003	10001	100	10001	Secret Garden	99999	30000
50003	30005	10001	1000	10001	Secret Garden	99999	30000
50005	30007	10005	1500	10005	Happy Bear	99995	23000
50007	30005	10010	600	10010	New town	99990	14000
50010	30002	10009	1200	10009	Boo King	99991	37000
50002	30001	10002	150	10002	Lost Horizon	99998	20000
50006	30001	10006	1000	10006	Scarry World	99994	12000

book Table을 구축입력으로 했을 때 조인결과이다. Test1과 출력순서는 좀 변했지만 전체적인 결과는 동일하다. 따라서 해쉬 인덱스가 테이블의 종류에 상관없이 올바르게 생성됨을 확인 할 수 있다.

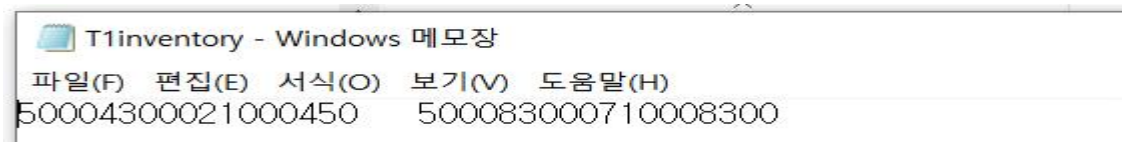
4) 임시 파일

- T0 (partition 0번 임시 파일)



T0book에는 book_id가 10003, 10007인 레코드가 T0inventory에는 book_id가 10003인 레코드가 들어있다. partition의 크기가 4 이므로 10003과 10007이 같은 곳에 들어가는 것은 옳다. inventory에는 book_id가 10003인 record만 존재하므로 올바르게 저장되었다.

-T1 (partition 1번 임시 파일)



T1book에는 book_id가 10004, 10008인 레코드가 T1inventory에는 book_id가 10004, 10008인 레코드가 들어가 있다. 파일에 저장된 레코드들을 비교해 보면 올바르게 모두 저장된 것을 확인 할 수 있다.

-T2 (partition 2번 임시 파일)

T2book에는 book_id가 10001, 10005, 10009 ,10010인 레코드가 T2inventory에는 book_id가 10001, 10005, 10009, 10010인 레코드가 들어가 있다. 각 문자의 아스키코드의 합을 partition 크기로 모듈러 해주는 것이기 때문에 10001과 10010은 같은 곳에 저장되는 것이 옳다. ('1'+'0'+'0'+'0'+'1' == '1'+'0'+'0'+'1'+'0' 이므로)

T2book의 경우 총 4개의 레코드가 저장되어 두 블록으로 나누어져 저장 되어 있는 것을 확인할 수 있다. (각 레코드가 30bytes 이므로 한 블록에 최대 3개의 레코드가 저장 가능한 상황)
inventory Table에서 book_id가 10001인 두 레코드 모두가 저장되어 있는 것을 확인 할 수 있다. 위의 레코드들을 파일에 저장된 레코드들과 비교해 보면 올바르게 모두 저장된 것을 확인 할 수 있다.

-T3 (partition 3번 임시 파일)

T3book에는 book_id가 10002, 10006인 레코드가 T3inventory에는 book_id가 10002, 10006인 레코드가 들어가 있다. 파일에 저장된 레코드들을 비교해 보면 올바르게 모두 저장된 것을 확인 할 수 있다.

T1~T3의 레코드 개수를 세어보면 각각 10개의 record가 저장되어 있고 각 레코드들을 모아서 비교해보면 원본 파일에 저장되어있는 테이블과 동일하다.

위와 같은 임시 파일 상황과 test1 test2가 모두 올바르게 동작하였으므로 해시 조인이 올바르게 동작하고 있음을 확인 할 수 있다.