2022-01 데이터베이스 시스템 설계 과제

관계 DB 저장 시스템 개발

Slotted Page와 가변 길이 레코드 포맷 구조를 중심으로

컴퓨터공학부 소프트웨어전공

20172848 정석우

목차

1. 개요

가변 길이 레코드 포맷과 Slotted Page Structure에 기반한 Relational Database System 설계 및 TDD를 적용한 구현

1-1) 요구사항 Specification

1. 시스템은 CLI 기반 어플리케이션으로써 동작해야 한다.
2. 시스템은 다른 응용에서 데이터 저장 및 조회를 위한 API를 제공해야 한다.
3. 시스템의 데이터는 파일에 저장되어야 하고, 시스템이 재시작, 종료, 초기화 되어도 파일에 있는 데이터의 무결성 (Persistence)는 보장되어야 한다.
4. 시스템 내부에서 태이블 생성 및 레코드 삽입이 가능해야 한다.
5. 시스템에 입력되는 데이터는 VARCHAR(N), 실제 프로그램 상에서는 std::string 또는 const char* 만을 대상으로 한다.
6. 시스템 내부에서 테이블의 Primary Key 값으로 레코드 검색이 가능해야 한다.
7. 시스템 내부에서 테이블의 컬럼 목록을 조회할 수 있어야 한다.

2. 구현 언어 및 개발 환경

구현 언어으로는 표준 C++20 을 사용하였다. 개발 환경은 Windows 10 Home (Build 19043.1706) 운영체제와 Visual Studio 2022 Community 버전을 사용하여 개발하였다.

3. 설계 설명

본 프로젝트의 설계는 크게 7 부분으로 나누어져 있다. 이를 top-down 방식으로 꾸려 나가면 아래와 같은 순서이다.

(1) 시스템 UI 모듈

(2) 시스템 메인 모듈

(3) 테이블 모듈

(4) 페이지 모듈

(5) 레코드 모듈

(6) 바이트 변환 모듈

(7) 메타 데이터의 json 형태 저장을 위한 외부 라이브러리 모듈 (jsoncpp)

3-1. 시스템 UI 모듈

시스템 UI 모듈에서는 main() 함수가 포함되어 있는 유저 CLI 환경을 제공한다. 시스템이 맨 처음 기동되어 메타 데이터를 읽고, 사용자로부터 입력 값을 받아 이를 레코드로 변환하고, 페이지 단위로 파일에 저장하도록 하는 가장 상위의 모듈이다. 시스템 UI 모듈에서는 후술할 시스템 메인 모듈을 초기화 및 사용하여 데이터의 읽기와 쓰기를 총괄하여 시행한다.

포함하고 있는 함수는 하나로, CLI 환경에서 사용자의 메뉴 선택을 돕기 위한 show_menu() 함수가 있다. show_menu() 함수는 단순히 메뉴를 출력하는 역할 그 이상도 이하도 수행하지 않는 매우 간단한 함수이다.

3-2. 시스템 메인 모듈

```cpp
#pragma once
#include <vector>
#include <string>
#include "table.h"
#include "jsoncpp/json/json.h"

#define FILE_MAX_BLOCK_NUM 1024
#define META_DATA_FILE_NAME "meta_data.json"

class SystemModule
{
private:
    std::vector<Table> table_list;
    block_store_loc next_block_ptr;
    Json::Value system_meta_json;
    std::string next_insert_file_name;
public:
    std::map<std::string, int> table_name_index_map;
    SystemModule(); // 메타 데이터 로드 및 시스템 초기 구동
    int insert_new_table(Table new_table); // 빈 테이블 생성 및 메타 데이터 업데이트
    int insert_new_record(int table_idx); // 테이블에 레코드 삽입
    int search_by_pk(int table_idx, std::string key);
    int get_table_column_list(int table_idx);
    int get_table_every_data(int table_idx);
    table_meta_data convert_json_to_meta(Json::Value data);
    Json::Value convert_meta_to_json(table_meta_data meta);
    int write_meta_data_to_file();
    void get_table_name_list();
};
```

시스템 메인 모듈은 데이터를 파일에 쓰고, 파일로부터 읽어오도록 하는 핵심 기능을 담당하는 역할이다. 데이터는 실제로 페이지 (Block) 단위로 쓰고 읽게 되지만 사용자의 데이터 입력과 출력은 물리적인 페이지 단위가 아닌 테이블 단위로 이루어지게 되기 때문에 그 설계 간극을 메우기 위한 모듈로써 생기게 되었다.

시스템 메인 모듈은 데이터를 논리적으로 저장하는 가장 큰 단위인 테이블 모듈의 목록을 가진다. 시스템 UI 모듈에서 가장 먼저 이루어지는 행위는 시스템 메인 모듈을 초기화 할 때 메타 데이터를 저장하는 json 파일에서 메타 데이터를 읽고 이를 파싱하여 논리적인 테이블의 목록을 구성하는 것이다. 이 때 파싱되는 데이터의 목록은 다음과 같다.

	키	값 설명	값 예시
1	FILE_MAX_BLOCK_NUM	파일에 저장되는 블록의 최대 크기	1024
2	next_insert_file_name	다음에 삽입될 데이터 파일의 이름	“data.db”
3	table_meta_data	테이블들의 메타 데이터	List
3-1	ㄴ table_name	테이블의 이름	“student”
3-2	ㄴ table_column_list	테이블의 컬럼 이름 목록	[“id”, “name”, “grade”]
3-3	ㄴ fixed_column_cnt	테이블의 고정 길이 칼럼 개수	2
3-4	ㄴ variable_column_cnt	테이블의 가변 길이 칼럼 개수	1
3-5	ㄴ fixed_column_length	테이블의 고정 길이 칼럼 길이 목록	[5, 1]
3-6	ㄴ pk_column_idx	테이블의 Primary Key 인덱스	0
3-7	ㄴ block_location	테이블의 데이터 저장 위치 목록	List

3-7-1	ㄴㄴ file_name	데이터가 저장된 파일 이름	“data.db"
3-7-2	ㄴㄴ start_loc	데이터가 저장된 파일의 시작 위치	0
3-7-3	ㄴㄴ end_loc	데이터가 저장된 파일의 끝 위치	4096

시스템 메인 모듈이 시스템 메타 데이터를 json 파일로부터 읽은 다음에는 즉시 테이블 목록에 테이블 메타 데이터를 할당한다. 이 때는 아직 테이블의 레코드 목록이 채워지지 않은 상태이며, 실제로 데이터가 담기게 되는 테이블의 레코드 목록은 상위 모듈인 시스템 UI 모듈에서 데이터 읽기와 쓰기를 요청했을 때 파일로부터 데이터를 읽어 테이블 목록에 레코드 목록을 채워 넣도록 설계하였다.

시스템 메인 모듈에 적재되는 데이터는 크게 4 종류이다.

첫 번째로 앞서 설명했듯이 실제 데이터가 담기는 공간인 논리적 구조인 테이블 목록인 table_list, 둘째는 시스템의 전체 메타 데이터를 Json 구조로 저장하고 있는 system_meta_json, 셋째는 다음 쓰기 대상이 되는 파일의 이름인 next_insert_file_name, 마지막으로 테이블의 이름으로부터 테이블 배열의 인덱스를 변환할 수 있도록 사용되는 table_name_index_map 이 있다.

데이터가 저장되는 파일의 입출력은 Block 단위로 시행된다. 본 시스템에서는 Block 의 크기는 4096 Byte 로 고정하여 진행하였다. 또한 파일에 들어가는 블록의 최대 크기인 FILE_MAX_BLOCK_NUM 은 1024 개로 고정하였다. 따라서 파일의 최대 크기는 4096 * 1024 Byte = 약 4MB 이다.

3-3. 테이블 모듈

```
#pragma once
#include <vector>
#include "record.h"
#include "meta_data.h"

typedef struct record_store_loc
{
    std::string file_name;
    int start_loc;
    int end_loc;
}record_store_loc;


class Table
{
private:
    column_info column_meta;
    table_meta_data table_meta;
    std::vector<Record> record_list;
    std::vector<record_store_loc> record_loc_list;
public:
    Table(table_meta_data table);
    Table(column_info column, std::vector<record_store_loc> record_loc_list);
    int load_from_file_location();
    int insert_new_record(std::string file_name, int start, int end);
    Record search_by_id(std::string search_key);
    std::vector<Record> get_record_list();
    table_meta_data get_table_meta();
    column_info get_column_meta();
    int insert_new_record_loc(record_store_loc new_location);
};
```

앞서 설명한 시스템 메인 모듈에서 테이블의 목록을 저장했는데, 바로 그 테이블의 구조는 위와 같이 설계하였다. 테이블 모듈은 논리적인 데이터 구조를 저장함으로써 상위 모듈인 시스템 메인 모듈에서 CLI 를 통해 메모리에 적재된 데이터를 사용자가 직관적으로 사용할 수 있도록 하는 데에 목적이 있다.

그렇기 때문에 테이블 모듈은 물리적인 데이터의 가장 큰 단위인 Page 를 저장하지 않는다. 대신 실제로 데이터가 들어있으며 구분 가능한 최소 데이터의 단위인 Record 의 목록을 저장한다.

3-4. 페이지 모듈

```
#pragma once
#include "record.h"
#include <vector>
#include "meta_data.h"

#define PAGE_SIZE 4096

//페이지 자체 메타 데이터, 페이지 크기, 레코드 개수, free space 마지막 위치, 컬럼 정보
typedef struct page_meta_data
{
    int entry_size;
    int free_space_end_addr;
    column_info column_meta;
}page_meta_data;

//페이지 내에 저장된 레코드의 메타 데이터
typedef struct record_meta_data
{
    int offset;
    int length;
    bool is_deleted;
}record_meta_data;

class SlottedPage
{
private:
    int page_idx; //page_idx * (PAGE_SIZE) ~ page_idx * (PAGE_SIZE + 1) 까지
    std::string file_name;
    page_meta_data meta_data;
    std::vector<record_meta_data> record_ptr_arr;
    std::vector<Record> record_arr;
public:
    SlottedPage(std::string file_name, column_info column_meta, int page_start, int page_end);
    SlottedPage(std::string file_name, column_info column_meta_info, int page_idx);
    std::vector<Record> get_record_list();
    void print_slotted_page();
    int write_page_on_disk();
    int add_record(Record tRecord);
    bool is_able_insert();
};
```

페이지 모듈은 가장 큰 데이터의 물리적 저장 단위이다. 페이지에는 크게 3 가지 중요한 데이터가 저장된다. 1. 페이지 각각의 메타 데이터와 2. 레코드 데이터 배열 3. 개별 레코드가 파일의 어느 위치에 저장되어 있는지, 삭제된 레코드인지를 저장하는 데이터 배열.

페이지의 생성은 크게 두 가지 방법으로 이루어진다. 첫째로는 사용자로 입력 받은 데이터로부터 새 페이지를 생성하는 방법, 또 한 가지는 파일에 입력되어 있는 데이터로부터 기존에 존재하던 페이지를 불러와 메모리에 적재하는 방법이다. 따라서 입력 파라미터가 다른 서로 다른 생성자를 오버로딩하여 모듈을 구성하였다.

페이지의 메타 데이터 안에는 페이지에 들어 있는 레코드 개수, 페이지의 Free space 가 끝나는 위치를 저장하며 이 두 가지 데이터는 페이지가 실제로 블록 단위로 파일에 쓰여질 때 가장 앞부분에 작성된다. 이는 int 형 데이터 두 개로 구성되므로 페이지의 가장 앞부분 8 byte 는 페이지 공통 메타 데이터로 저장되는 셈이다. 그 이후에는 개별 레코드가 위치한 주소인 offset, 개별 레코드의 길이인 length, 삭제 여부인 is_deleted 가 저장된다. 따라서 페이지에 저장된 레코드 당 총 9 byte 만큼의 개별 레코드에 대한 메타 데이터 공간을 필요로 한다. 이를 그림으로 그려보면 아래와 같다.

A : 페이지 공통 데이터. 페이지 시작부터 8 Byte

B: 레코드 고유 메타 데이터. 각각 9 Byte 고정 (offset, length, is_deleted)

- B-1 은 1 번 레코드 메타 데이터, B-2 는 2 번 레코드 메타 데이터이다.

C : 페이지의 Free space. 실제로는 0x00 (0, null)로 차 있다.

D : 실제 데이터가 저장되어 있는 가변 길이 레코드. 가변 길이 레코드의 구조는 이후 레코드 모듈에서 후술한다.

- D-1 은 1 번 레코드, D-2 는 2 번 레코드 데이터이다. 페이지의 끝부터 차례로 Free space 를 채워가는 역순으로 레코드가 더해진다.

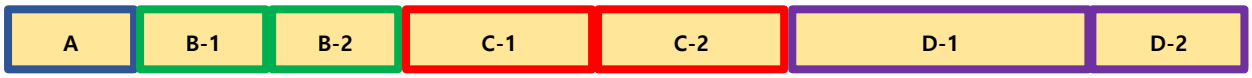
사용자로부터 입력 받은 데이터가 페이지에 저장된 이후 실제로 파일에 쓰이게 될 때에는 후술할 바이트 변환 모듈을 사용하여 바이트 배열로 변환이 이루어진 이후 파일의 시작 주소부터 끝 주소까지 기록되는 방식으로 설계하였다.

3-5. 레코드 모듈

```
#pragma once
#include <vector>
#include <string>
#include "meta_data.h"

#define MAX_COLUMN_LEN 50

typedef struct location_meta_data
{
    int offset;
    int length;
}location_meta_data;


class Record
{
private:
    unsigned int null_bitmap;
    std::vector<std::pair<std::string, int>> fixed_len_column;
    std::vector<std::string> var_len_column;
    std::vector<location_meta_data> var_len_column_loc;
    std::vector<unsigned char> record_byte_arr;
    int byte_arr_size;
public:
    Record();
    Record(unsigned char* byte_arr, int arr_length, column_info column_meta);
    Record(std::vector<std::string> input, column_info column_meta);
    int get_record_size();
    void print_record();
    int get_null_bitmap();
    std::vector<std::string> get_fixed_column_list();
    std::vector<std::string> get_var_column_list();
    std::string to_string();
    std::vector<unsigned char> to_byte_vector();
};
```

본 프로젝트에서 레코드 모듈은 가변 길이 레코드 구조를 가진다. 가변 길이 레코드 구조란, 가변 길이의 컬럼 데이터를 저장하기 위해 고안된 구조이다. 기본적으로 고정 길이 컬럼과 가변 길이 칼럼은 다른 속성을 지니기 때문에 실제 물리적으로 파일에 기록될 때에도 같은 레코드 내에서도 구분되어져야 한다.

따라서 이 프로젝트에서는 레코드 구조는 아래 그림과 같이 설계되었다.



A : Null bitmap. 이는 4 byte (32 bit) 로 고정된 크기를 가지기 때문에 레코드 가질 수 있는 총 컬럼 개수는 32 개로 고정이다. 0 번째 컬럼은 0x00000001 에 해당하는 null bitmap 을 가진다. 다르게 말하면 Null bitmap 과 0x00000001 을 bitwise AND 연산하였을 때 true 이면 0 번째 컬럼은 Null 값을 가지며, 컬럼을 읽을 때 해당 컬럼은 제외하고 읽는다.

B : 고정 길이 컬럼의 값이다. 고정 길이 컬럼의 최대 길이는 상위 모듈인 테이블에 저장되어 있기 때문에 이를 참고하여 해당 값을 읽어온다. B-1 은 첫 번째 고정 길이 칼럼, B-2 는 두 번째 고정 길이 칼럼의 데이터를 저장한다. 고정 길이 칼럼의 데이터 자료형은 unsigned char 이다. (1 byte, 0~255)

C : 가변 길이 칼럼이 저장되어 있는 위치를 알린다. 둘 다 int 자료형인 offset, length 가 들어 있는 구조체로써 메모리에 적재되기 때문에 파일에 쓰여질 때에도 각각의 가변 길이 칼럼 위치는 8 byte 의

크기를 가진다. C-1 은 첫 번째 가변 길이 칼럼이 저장된 offset과 첫 번째 가변 길이 칼럼의 길이인 length, C-2 는 두 번째 가변 길이 칼럼이 저장된 offset 과 그 길이인 length 를 저장한다. 즉 이 예시에서 이 레코드를 저장하고 있는 테이블은 총 2 개의 고정 길이 칼럼과 2 개의 가변 길이 칼럼을 가진다는 것을 알 수 있다

D : 가변 길이 칼럼의 데이터가 저장된다. 가변 길이 칼럼의 데이터 자료형은 고정 길이 칼럼과 마찬가지로 unsigned char 자료형으로 고정된다. 각각의 문자들은 아스키 코드 값으로써 이진 파일 데이터에 기록된다. D-1 은 첫 번째 가변 길이 칼럼의 데이터, D-2 는 두 번째 가변 길이 칼럼의 데이터를 나타낸다.

레코드가 실제로 파일에 저장될 때 가변 길이 레코드 포맷을 가지는 것 외에도, 논리적인 레코드 모듈의 구조에서는 각각의 데이터를 std::string 자료형으로 저장하는 배열이 있다. 메모리에 적재된 레코드들은 다시 파일에 쓰여질 때 레코드 모듈의 to_byte_vector() 함수를 통해 unsigned char 배열로 변환된다. 변환된 배열은 다시 상위 모듈인 페이지에서 실제 파일에 쓰여지는 행위를 통해 최종적으로 파일에 기록된다.

3-6. 바이트 변환 모듈

```
#pragma once
#include <vector>

std::vector<unsigned char> int_to_byte(int x);
int byte_arr_to_int(unsigned char* x);
void print_byte_arr(unsigned char* byte_arr, int arr_length);
```

바이트 변환 모듈은 메모리에 적재되어 있는 데이터를 실제 파일에 기록하고, 파일에 기록되어 있는 데이터를 다시 메모리로 읽어 들이는 과정에서 변환 과정이 필요한 몇 가지 유용한 함수들을 포함한다.

3-7. 메타 데이터의 json 형태 저장을 위한 외부 라이브러리 모듈 (jsoncpp)

본 프로젝트에서는 시스템 메타 데이터를 json 파일로써 작성하고 또 업데이트하기 때문에 외부 라이브러리를 사용하여 그 생산성을 높이고자 하였다. 이에 C++ 개발 분야에서 가장 널리 사용되는 json 라이브러리인 jsoncpp 라이브러리를 사용하였다.

4. 구현 설명

시스템 UI 모듈에서 실행되는 6가지 주요 기능들이 어떻게 구현되었는지 서술하겠다.

1) 시스템 로드 (시스템 메인 모듈 초기화)

2) 테이블 삽입 (CREATE TABLE)

3) 테이블에 레코드 삽입 (INSERT INTO TABLE)

4) 테이블의 모든 데이터 조회 (SELECT FROM TABLE)

5) 테이블의 PK 값으로 특정 데이터 조회 (SELECT FROM TABLE WHERE PK = ?)

6) 테이블의 컬럼 목록 조회 (SELECT COLUMN NAME FROM TABLE)

4-1. 시스템 로드 (시스템 메인 모듈 초기화)

시스템의 메타 데이터는 특정 이름을 가진 json 파일로써 저장된다. 이에 최상위 모듈인 시스템 UI 모듈에서는 하위 모듈인 시스템 메인 모듈을 초기화하여 메타 데이터를 읽어오고, 테이블의 목록과 시스템 운용에 필수적인 데이터들을 메모리에 적재한다. 이는 SystemModule() 생성자로 이루어진다.

```
std::fstream meta_file;
meta_file.open(META_DATA_FILE_NAME, std::ios::binary | std::ios::in);
std::string str;
std::cout << "Reading meta data file.." << std::endl;
if (meta_file.is_open())
{
    while (!meta_file.eof())
    {
        std::string tmp;
        std::getline(meta_file, tmp);
        str.append(tmp);
    }
}
meta_file.close();

//json 파일 파싱하여 Table 데이터 리스트화
//Table 데이터는 바로 read 하지 않고 나중에 메모리로 올림
Json::Reader reader;
Json::Value read_data;
if (!reader.parse(str, read_data))
{
    std::cerr << "Failed to parse json : " << reader.getFormattedErrorMessages() << std::endl;
}
system_meta_json = read_data;
```

위 코드는 메타 데이터 파일을 읽어와 이를 jsoncpp 라이브러리의 파싱 기능을 사용하여 시스템 메인 모듈 private 변수인 Json::Value 자료형 system_meta_json에 복사하는 코드이다.

```
//json 파일 파싱하여 Table 데이터 리스트화
//Table 데이터는 바로 read 하지 않고 나중에 메모리로 올림
Json::Reader reader;
Json::Value read_data;
if (!reader.parse(str, read_data))
{
    std::cerr << "Failed to parse json : " << reader.getFormattedErrorMessages() << std::endl;
}
system_meta_json = read_data;
next_insert_file_name = read_data["next_insert_file_name"].asCString();
Json::Value table_meta = read_data["table_meta_data"];
int cnt = 0;
for (auto i = table_meta.begin(); i != table_meta.end(); i++)
{
    std::string table_name = (*i)["table_name"].asCString();
    Json::Value block_location_json = (*i)["block_location"];
    std::vector<block_store_loc> block_location;
    for (auto it = block_location_json.begin(); it != block_location_json.end(); it++)
    {
        std::string file_name = (*it)["file_name"].asCString();
        int start_loc = (*it)["start_loc"].asInt();
        int end_loc = (*it)["end_loc"].asInt();
        block_store_loc each_block_info = block_store_loc{file_name, start_loc, end_loc};
        block_location.push_back(each_block_info);
    }
    std::vector<std::string> table_column_list;
    for (auto it = (*i)["table_column_list"].begin(); it != (*i)["table_column_list"].end(); it++)
    {
        table_column_list.push_back((*it).asCString());
    }

    std::vector<int> fixed_column_length;
    for (auto it = (*i)["fixed_column_length"].begin(); it != (*i)["fixed_column_length"].end(); it++)
    {
        fixed_column_length.push_back((*it).asInt());
    }
    int variable_column_cnt = (*i)["variable_column_cnt"].asInt();
    int fixed_column_cnt = (*i)["fixed_column_cnt"].asInt();
    int pk_column_idx = (*i)["pk_column_idx"].asInt();
    table_meta_data tmd = table_meta_data{table_name, block_location, table_column_list, fixed_column_length, variable_column_cnt, fixed_column_cnt, pk_column_idx};
    Table each_table = Table(tmd);
    table_list.push_back(each_table);
    table_name_index_map.insert({ table_name, cnt });
    cnt++;
}
std::cout << "Finished loading total " + std::to_string(table_list.size()) + " table from meta data." << std::endl << "Table List :: ";
for (int i = 0; i < table_list.size(); i++)
{
    std::cout << table_list[i].get_table_meta().table_name << ", ";
}
std::cout << std::endl;
```

위 코드는 파일로부터 읽어 들인 메타 데이터를 차례로 구문 분석하여 각 테이블의 메타 데이터로써 변환하여 테이블의 목록으로써 다시 메모리에 적재하는 과정이다. 테이블의 메타 데이터만 로드하기 때문에 실제 저장되어 있는 데이터는 이 과정에서 로드되지 않는다.

4-2. 테이블 삽입

테이블의 메타 데이터는 메타 데이터 파일에 저장되기 때문에 테이블이 삽입된다는 것은 곧 시스템 고유 메타 데이터 파일을 업데이트 하는 일련의 과정과 동일하다.

```
int SystemModule::insert_new_table(Table new_table)
{
    {
        table_meta_data table_meta = new_table.get_table_meta();
        Json::Value write_data = convert_meta_to_json(table_meta);
        system_meta_json["table_meta_data"].append(write_data);
        table_list.push_back(new_table);
        table_name_index_map[new_table.get_table_meta().table_name] = table_name_index_map.size();
        write_meta_data_to_file();
    }
    return 0;
}
```

시스템 UI 모듈이 사용자로부터 테이블 생성에 필요한 값들을 전부 입력 받고 난 후에는 이를 메타 데이터로 변환시켜 새 테이블 객체를 생성한다. 이후 이 테이블 객체는 다시 메타 데이터 Json 객체로 변환되어 메모리에 적재되어 있는 기존 메타 데이터를 덮어씌워 업데이트한다. 업데이트 된 즉시 새로 생긴 테이블의 메타 데이터를 메타 데이터 파일에 기록하여 데이터가 손실되는 경우를 줄이도록 하였다.

```cpp
int SystemModule::write_meta_data_to_file()
{
    Json::StreamWriterBuilder builder;
    builder["commentStyle"] = "None";
    builder["indentation"] = "    ";

    std::unique_ptr<Json::StreamWriter> writer(builder.newStreamWriter());
    std::ofstream outputFileStream(META_DATA_FILE_NAME);
    writer->write(system_meta_json, &outputFileStream);
    return 0;
}
```

파일에 메타 데이터를 업데이트 하기 위해서는 write_meta_data_to_file() 함수를 사용한다. 이 함수에서는 jsoncpp 라이브러리의 StreamWriteBuilder 빌더 객체를 사용하여 출력 파일 스트림에 기록하고자 하는 데이터 (이 경우에는 메모리에 적재되어 있는 메타 데이터 json 객체)를 파일에 기록하는 식으로 구현하였다.

```cpp
string table_name;
cin >> table_name;
if (system_module.table_name_index_map.count(table_name) != 0)
{
    cout << "Table with same name : " << table_name << " already exists!" << endl;
    system_module.get_table_name_list();
    break;
}
```

시스템 메인 모듈에는 테이블의 이름과 그 테이블의 배열 인덱스 쌍이 저장되어 있는 테이블 메타 데이터가 std::map<std::string, int> 자료형으로 존재한다. 때문에 신규 테이블을 생성할 시에 이름이 동일한 테이블이 있을 경우 이를 감지하고 사용자에게 이미 같은 이름의 테이블이 있음을 알려주도록 구현하였다.

4-3. 테이블에 레코드 삽입

테이블에 새로운 레코드를 삽입하는 함수인 insert_new_record() 함수는 크게 네 부분으로 나누어진다. 첫 번째 부분은 사용자로부터 새롭게 생성할 테이블 정보를 입력 받는 부분이다.

```cpp
int SystemModule::insert_new_record(int table_idx)
{
    Table target_table = table_list[table_idx];
    //target_table.load_from_file_location();
    std::vector<std::string> column_name_list = target_table.get_table_meta().table_column_list;
    std::vector<std::string> new_column_value;
    std::cin.ignore();
    for (int i = 0; i < column_name_list.size(); i++)
    {
        std::string input;

        while (true)
        {
            input = "";
            std::cout << "Insert value for column name " << column_name_list[i] << " : ";
            std::getline(std::cin, input);
            if (i < target_table.get_table_meta().fixed_column_cnt)
            {
                if (input.length() > target_table.get_table_meta().fixed_column_length[i])
                {
                    std::cout << "Column Length Exceeded. Please input less than " << target_table.get_table_meta().fixed_column_length[i] << std::endl;
                }
                else break;
            }
            else break;
        }
        new_column_value.push_back(input);
    }
```

테이블의 모든 컬럼은 물리적으로 저장될 때 가변 길이 레코드 포맷으로 변환되어 저장되기 때문에 논리적인 구조 또한 고정 길이 컬럼 이후에 가변 길이 컬럼이 위치하도록 구현하였다. 때문에 고정 길이 컬럼의 길이를 초과하는 값이 고정 길이 컬럼의 값으로 들어왔을 때 예외 처리를 위와 같은 방식으로 구현하였다.

테이블에 새롭게 레코드를 삽입할 때, 기존 파일과 블록과의 상관 관계를 생각해보면 총 세 가지 경우가 가능했다.

첫 번째는 테이블이 저장되어 있는 같은 파일의 같은 블록에 레코드를 삽입할 수 있는 경우,

두 번째는 테이블이 저장되어 있는 기존 블록들이 꽉 찼기 때문에 같은 파일의 신규 블록으로 레코드를 삽입할 수 있는 경우,

세 번째는 테이블이 저장되어 있는 기존 블록도 전부 찼고, 같은 파일에도 더 이상 블록을 생성할 수 없어 신규 파일의 신규 블록으로 레코드를 삽입하는 경우이다.

```
Record new_record = Record(new_column_value, target_table.get_column_meta());
std::vector<block_store_loc> target_block_location = target_table.get_table_meta().block_location;
bool insert_flag = false;
std::string update_file_name = next_insert_file_name;
for (int i = 0; i < target_block_location.size(); i++)
{
    std::string file_name = target_block_location[i].file_name;
    update_file_name = file_name;
    int start = target_block_location[i].start_loc;
    int end = target_block_location[i].end_loc;
    SlottedPage page_read_from_disk = SlottedPage(file_name, target_table.get_column_meta(), start, end);
    //기존 page 내에 record 삽입 가능한 경우
    if (page_read_from_disk.is_able_insert())
    {
        page_read_from_disk.add_record(new_record);
        insert_flag = true;
        page_read_from_disk.write_page_on_disk();
        break;
    }
}
```

먼저 첫 번째 경우, 테이블이 저장되어 있는 같은 파일의 같은 블록에 레코드를 삽입할 수 있는 경우이다. 블록 단위로 페이지를 읽어왔는데 해당 페이지의 free space 길이가 새롭게 생성한 레코드의 길이보다 커서 기존 페이지의 free space에 레코드를 삽입할 수 있는 경우이다. 테이블 메타 데이터로부터 기존 파일과 블록 위치를 불러온 후 이를 페이지로 메모리에 적재한다. 그 이후 삽입이 가능한지 is_able_insert() 함수로 판별한 후 페이지에 레코드를 삽입한 후 다시 파일에 쓴다. 이 때 insert_flag를 true로 만들어 이후 진행될 분기는 skip 하도록 처리하였다.

```
int current_file_size = std::filesystem::file_size(update_file_name);

if (current_file_size < FILE_MAX_BLOCK_NUM * PAGE_SIZE)
{
    // 기존 파일에 block만 추가하는 경우
    int current_file_block_idx = current_file_size / PAGE_SIZE;

    int new_start_loc = (current_file_block_idx)*PAGE_SIZE;
    int new_end_loc = (current_file_block_idx + 1) * PAGE_SIZE;

    SlottedPage new_page_to_disk = SlottedPage(update_file_name, target_table.get_column_meta(), current_file_block_idx);

    new_page_to_disk.add_record(new_record);
    new_page_to_disk.write_page_on_disk();

    std::string table_name = target_table.get_table_meta().table_name;

    table_list[table_idx].get_table_meta().block_location.push_back(block_store_loc{update_file_name, new_start_loc, new_end_loc});
    table_list[table_idx].insert_new_record_loc(record_store_loc{update_file_name, new_start_loc, new_end_loc});

    for (auto it = system_meta_json["table_meta_data"].begin(); it != system_meta_json["table_meta_data"].end(); it++)
    {
        if (table_name.compare((*it)["table_name"].asCString()) == 0)
        {
            Json::Value new_block_location;
            new_block_location["file_name"] = update_file_name;
            new_block_location["start_loc"] = new_start_loc;
            new_block_location["end_loc"] = new_end_loc;
            (*it)["block_location"].append(new_block_location);

            write_meta_data_to_file();
            break;
        }
    }
}
```

두 번째 경우이다. 같은 파일의 같은 블록이 꽉 찼기 때문에 (free space의 크기가 신규 레코드의 크기보다 작은 경우) 같은 파일의 신규 블록에 신규 레코드를 기록한다. 이 경우 신규 레코드를 신규 블록에 쓴 이후에 테이블의 레코드들이 저장된 위치 배열 메타 데이터를 업데이트 해줘야 한다. 시스템 메타 데이터를 순회하며 같은 이름의 테이블이 나타날 때 테이블의 레코드들이 저장되어 있는 위치 배열인 block_location 항목에 신규 블록의 파일 이름, 블록 시작 위치와 끝 위치를 업데이트한 이후에 메타 데이터 파일도 업데이트 해준다.

```
else
{
    // 기존 파일이 꽉 차서 새 파일을 써야하는 경우
    size_t name_prefix = next_insert_file_name.find(".");
    if (name_prefix != std::string::npos)
    {
        name_prefix += 2;
        std::string file_number = next_insert_file_name.substr(name_prefix);
        int file_number_int = 0;
        if (file_number.compare(""))
        {
            file_number_int = 1;
        }
        else
        {
            file_number_int = std::stoi(file_number);
        }
        next_insert_file_name = "data" + std::to_string(file_number_int) + ".db";
    }
    else
    {
        // file name format error
    }
    std::string new_file_name = next_insert_file_name;
    SlottedPage new_page_to_disk = SlottedPage(new_file_name, target_table.get_column_meta(), 0);
    new_page_to_disk.add_record(new_record);
    new_page_to_disk.write_page_on_disk();

    std::string table_name = target_table.get_table_meta().table_name;
    system_meta_json["next_insert_file_name"] = next_insert_file_name;
    for (auto it = system_meta_json["table_meta_data"].begin(); it != system_meta_json["table_meta_data"].end(); it++)
    {
        if (table_name.compare((*it)["table_name"].asCString()) == 0)
        {
            Json::Value new_block_location;
            new_block_location["file_name"] = new_file_name;
            new_block_location["start_loc"] = 0;
            new_block_location["end_loc"] = PAGE_SIZE;
            (*it)["block_location"].append(new_block_location);

            write_meta_data_to_file();
            break;
        }
    }
    table_list[table_idx].get_table_meta().block_location.push_back(block_store_loc{ new_file_name, 0, PAGE_SIZE });
    table_list[table_idx].insert_new_record_loc(record_store_loc{ new_file_name, 0, PAGE_SIZE });
}
```

세 번째 경우이다. 기존 파일의 블록이 모두 꽉 찼고, 더 이상 같은 파일에 신규 블록을 생성할 수 없어 (최대 파일 크기 약 4MB) 신규 파일에 신규 블록으로 신규 레코드를 삽입해야 한다. 신규 파일의 이름의 형식은 data.db, data1.db, data2.db… 로 한다. 신규 파일의 이름과 파일의 어느 위치에 삽입될 지가 달라진 것 빼고는 두 번째 경우와 동일하다. 해당 위치에 레코드를 기입하고 메타 데이터와 그 파일을 업데이트 한다.

4-4. 테이블의 모든 데이터 조회

```
int SystemModule::get_table_every_data(int table_idx)
{
    Table target_table = table_list[table_idx];
    target_table.load_from_file_location();
    std::vector<Record> record_list = target_table.get_record_list();
    if (record_list.size() == 0)
    {
        std::cout << "table has no records. Please insert new record for table : " << target_table.get_table_meta().table_name << std::endl;
        return 0;
    }
    std::cout << "Printing total " + std::to_string(record_list.size()) + " records.." << std::endl;
    for (int i = 0; i < record_list.size(); i++)
    {
        std::cout << "Record index (" + std::to_string(i) + ")" << std::endl;
        record_list[i].print_record();
    }
    return 0;
}
```

테이블의 모든 데이터를 조회할 때에는 load_from_file_location() 함수로 조회할 테이블의 레코드 목록을 먼저 파일에서 메모리로 적재한다.

```
int Table::load_from_file_location()
{
    int file_num = record_loc_list.size();
    for (int i = 0; i < file_num; i++)
    {
        SlottedPage read_page = SlottedPage(record_loc_list[i].file_name, column_meta, record_loc_list[i].start_loc, record_loc_list[i].end_loc);
        std::vector<Record> record_list_from_page = read_page.get_record_list();
        for (int j = 0; j < record_list_from_page.size(); j++)
        {
            record_list.push_back(record_list_from_page[j]);
        }
    }
    return 0;
}
```

load_from_file_location() 함수는 테이블의 레코드 저장 위치 배열로부터 이를 페이지 단위로 읽어와 레코드 목록에 하나씩 쌓는 역할을 수행한다. 레코드 목록이 불러져 왔다는 것은 이미 테이블에 기록되어 있는 데이터가 전부 메모리에 적재되어 있다는 것이니 각각의 Record 객체의 print_record() 함수를 통해 출력함으로써 테이블의 모든 데이터를 출력하도록 구현하였다.

4-5. 테이블의 PK 값으로 특정 데이터 조회

```
int SystemModule::search_by_pk(int table_idx, std::string key)
{
    Table target_table = table_list[table_idx];
    target_table.load_from_file_location();
    std::vector<Record> record_list = target_table.get_record_list();
    int pk_idx = target_table.get_table_meta().pk_column_idx;
    for (int i = 0; i < record_list.size(); i++)
    {
        std::string search_value;
        if (pk_idx > target_table.get_table_meta().fixed_column_cnt)
        {
            search_value = record_list[i].get_var_column_list().at(pk_idx);
        }
        else
        {
            search_value = record_list[i].get_fixed_column_list().at(pk_idx);
        }
        if (search_value.compare(key) == 0)
        {
            record_list[i].print_record();
            return 0;
        }
    }
    std::cout << "Cannot find record with PK value = " << key << std::endl;
    return 0;
}
```

테이블의 PK 값으로 특정 데이터만 조회하는 함수는 상기 search_by_pk(int table_idx, std::string key) 함수로 구현하였다. 먼저 찾고자 하는 테이블을 '테이블의 모든 데이터를 조회'할 때처럼 페이지 단위로 파일에서부터 읽어 온 후에는 메모리에 레코드 목록이 모두 적재되어 있으므로 이후에는 각각의 레코드들의 pk 컬럼 값과 입력 받은 쿼리의 search key 값이 일치하는지 판별하여 일치하는 경우 해당 레코드를 출력하도록 하여 구현하였다. 만약 메모리에 적재되어 있는 레코드 중 search key 값과 일치하는 pk 값을 가진 레코드가 없을 경우 해당 테이블의 모든 레코드가 해당 값을 가지고 있지 않은 것이기 때문에 어떠한 레코드도 출력하지 않는다.

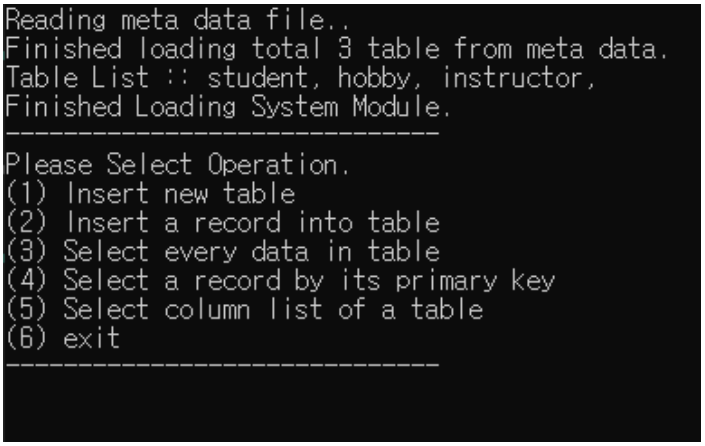
4-6. 테이블의 컬럼 목록 조회

```
int SystemModule::get_table_column_list(int table_idx)
{
    Table target_table = table_list[table_idx];
    std::vector<std::string> column_name_list = target_table.get_table_meta().table_column_list;
    int pk_idx = target_table.get_table_meta().pk_column_idx;
    for (int i = 0; i < column_name_list.size(); i++)
    {
        std::cout << "column " << std::to_string(i) << " : " << column_name_list[i];
        if (i == pk_idx)
        {
            std::cout << " (PK)" << std::endl;
        }
        else
        {
            std::cout << std::endl;
        }
    }
    return 0;
}
```

테이블의 컬럼 목록은 시스템 메인 모듈이 초기화 되면서 생성될 때 메타 데이터 파일로부터 메모리에 적재된다. 따라서 테이블의 컬럼 목록 조회는 단순히 메모리에 적재되어 있는 해당 테이블의 컬럼 목록을 출력하는 것만으로 간단히 구현하였다.

5. 기능 동작 정확성 검증 결과

5-1. 시스템 구동 시 시스템 메타 데이터 읽어 오는지 테스트



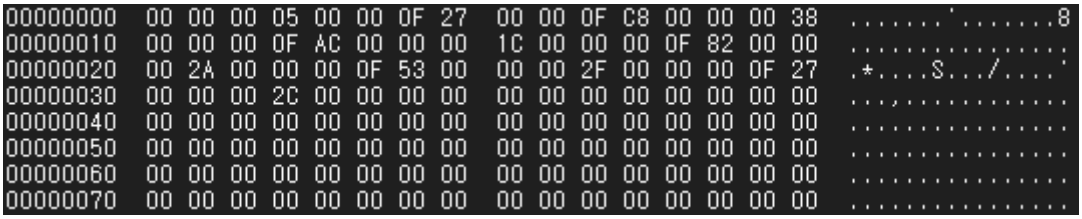
이 시점의 메타 데이터 파일 (json 파일)

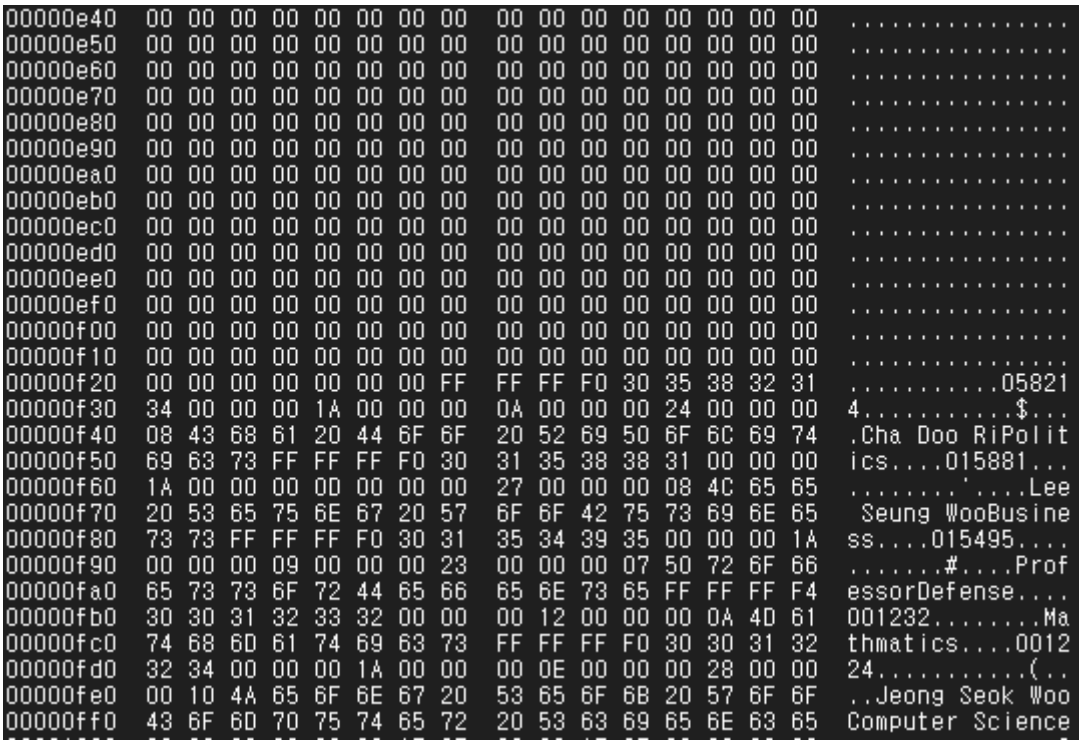

```
{
    "FILE_MAX_BLOCK_NUM" : 1024,
    "next_insert_file_name" : "data.db",
    "table_meta_data" :
    [
        {
            "block_location" :
            [
                {
                    "end_loc" : 4096,
                    "file_name" : "data.db",
                    "start_loc" : 0
                }
            ],
            "fixed_column_cnt" : 2,
            "fixed_column_length" : [ 5, 1 ],
            "pk_column_idx" : 0,
            "table_column_list" : [ "student_id", "grade", "name", "major" ],
            "table_name" : "student",
            "variable_column_cnt" : 2
        },
        {
            "block_location" :
            [
                {
                    "end_loc" : 12288,
                    "file_name" : "data.db",
                    "start_loc" : 8192
                }
            ],
            "fixed_column_cnt" : 1,
            "fixed_column_length" : [ 5 ],
            "pk_column_idx" : 0,
            "table_column_list" : [ "id", "name", "hobby" ],
            "table_name" : "hobby",
            "variable_column_cnt" : 2
        },
        {
            "block_location" :
            [
                {
                    "end_loc" : 16384,
                    "file_name" : "data.db",
                    "start_loc" : 12288
                }
            ],
            "fixed_column_cnt" : 1,
            "fixed_column_length" : [ 5 ],
            "pk_column_idx" : 0,
            "table_column_list" : [ "inst_id", "dept_name", "name" ],
            "table_name" : "instructor",
            "variable_column_cnt" : 2
        }
    ]
}
```

5-2. 테이블 데이터 조회 테스트

```
------------------------------
Please Select Operation.
(1) Insert new table
(2) Insert a record into table
(3) Select every data in table
(4) Select a record by its primary key
(5) Select column list of a table
(6) exit
------------------------------
3
Select * From Table
------------------------------
Please input table name to search every data : student
read starts from : 4040 to : 4096
read starts from : 4012 to : 4040
read starts from : 3970 to : 4012
read starts from : 3923 to : 3970
read starts from : 3879 to : 3923
Printing total 5 records..
Record index (0)
null bitmap : fffffff0
fixed_len_column :
00122
4
var_len_column :
Jeong Seok Woo
Computer Science

Record index (1)
null bitmap : fffffff4
fixed_len_column :
00123
2
var_len_column :
NULL
Mathmatics

Record index (2)
null bitmap : fffffff0
fixed_len_column :
01549
5
var_len_column :
Professor
Defense

Record index (3)
null bitmap : fffffff0
fixed_len_column :
01588
1
var_len_column :
Lee Seung Woo
Business

Record index (4)
null bitmap : fffffff0
fixed_len_column :
05821
4
var_len_column :
Cha Doo Ri
Politics
```

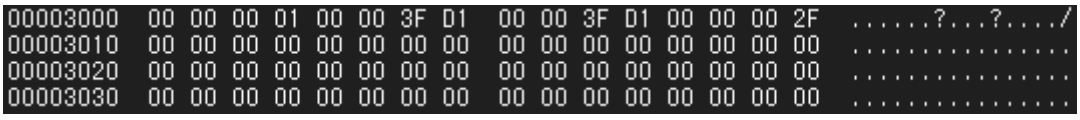
아래는 실제 데이터 파일에 쓰여져 있는 이진 데이터이다.





5-3. 테이블에 레코드 삽입 테스트

레코드 삽입 전



```
00003f70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003f80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003f90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003fa0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003fb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003fc0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003fd0  00 FF FF FF F8 30 30 30  30 34 00 00 00 19 00 00  .....00004......
00003fe0  00 0A 00 00 00 23 00 00  00 0C 6D 61 74 68 6D 61  .....#....mathma
00003ff0  74 69 63 73 48 61 72 72  79 20 50 6F 74 74 65 72  ticsHarry Potter
```

레코드 삽입

```
Insert Record To Table
------------------------------
Please input table name to insert record : instructor
Insert value for column name inst_id : 00005
Insert value for column name dept_name : art
Insert value for column name name : Picasso
read starts from : 4049 to : 4096
```

레코드 삽입 후

```
Select * From Table
------------------------------
Please input table name to search every data : instructor
read starts from : 4049 to : 4096
read starts from : 4014 to : 4049
Printing total 2 records..
Record index (0)
null bitmap : fffffff8
fixed_len_column :
00004
var_len_column :
mathmatics
Harry Potter

Record index (1)
null bitmap : fffffff8
fixed_len_column :
00005
var_len_column :
art
Picasso
```

```
00003000  00 00 00 02 00 00 3F AE  00 00 3F D1 00 00 00 2F  ......?...?..../
00003010  00 00 00 3F AE 00 00 00  23 00 00 00 00 00 00 00  ...?....#.......
00003020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
00003f70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003f80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003f90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00003fa0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 FF FF  ................
00003fb0  FF F8 30 30 30 30 35 00  00 00 19 00 00 00 03 00  ..00005.........
00003fc0  00 00 1C 00 00 00 07 61  72 74 50 69 63 61 73 73  .......artPicass
00003fd0  6F FF FF FF F8 30 30 30  30 34 00 00 00 19 00 00  o....00004......
00003fe0  00 0A 00 00 00 23 00 00  00 0C 6D 61 74 68 6D 61  .....#....mathma
00003ff0  74 69 63 73 48 61 72 72  79 20 50 6F 74 74 65 72  ticsHarry Potter
```

5-4. 레코드 pk 검색 테스트

테스트 대상 레코드 데이터

```
Select * From Table
------------------------------
Please input table name to search every data : instructor
read starts from : 4049 to : 4096
read starts from : 4014 to : 4049
Printing total 2 records..
Record index (0)
null bitmap : fffffff8
fixed_len_column :
00004
var_len_column :
mathmatics
Harry Potter

Record index (1)
null bitmap : fffffff8
fixed_len_column :
00005
var_len_column :
art
Picasso
```

레코드를 성공적으로 찾은 경우 일치하는 PK 값을 가진 레코드 출력

```
Select * From Table Where PK = ?
------------------------------
Please input table name to search by pk value : instructor
Please input value for search pk value : 00004
read starts from : 4049 to : 4096
read starts from : 4014 to : 4049
null bitmap : fffffff8
fixed_len_column :
00004
var_len_column :
mathmatics
Harry Potter
```

레코드를 찾지 못한 경우 (일치하는 PK 값이 없는 경우)

```
Select * From Table Where PK = ?
------------------------------
Please input table name to search by pk value : instructor
Please input value for search pk value : 00012
read starts from : 4049 to : 4096
read starts from : 4014 to : 4049
Cannot find record with PK value = 00012
```

5-5. 테이블 생성

```
Table Insert
----------------------------
Please input new table name : subject
How many fixed length columns will be in the new table? : 3
Please input name of index (0) fixed length column : subject_id
Please input length of index (0) fixed length column : 8
Please input name of index (1) fixed length column : credit
Please input length of index (1) fixed length column : 1
Please input name of index (2) fixed length column : semester
Please input length of index (2) fixed length column : 7
How many variable length columns will be in the new table? : 1
Please input name of index (0) variable length column : subject_name
Which index would be the primary column for the search? : 0
```

업데이트 된 시스템 메타 데이터 파일


```
{
    "block_location" :
    [
        {
            "end_loc" : 16384,
            "file_name" : "data.db",
            "start_loc" : 12288
        }
    ],
    "fixed_column_cnt" : 1,
    "fixed_column_length" : [ 5 ],
    "pk_column_idx" : 0,
    "table_column_list" : [ "inst_id", "dept_name", "name" ],
    "table_name" : "instructor",
    "variable_column_cnt" : 2
},
{
    "block_location" : [],
    "fixed_column_cnt" : 3,
    "fixed_column_length" : [ 8, 1, 7 ],
    "pk_column_idx" : 0,
    "table_column_list" : [ "subject_id", "credit", "semester", "subject_name" ],
    "table_name" : "subject",
    "variable_column_cnt" : 1
}
```


테이블 생성 직후 조회 결과

```
Select * From Table
----------------------------
Please input table name to search every data : subject
table has no records. Please insert new record for table : subject
----------------------------
```

같은 이름의 테이블로 생성 시도하였을 때

```
Table Insert
------------------------------
Please input new table name : subject
Table with same name : subject already exists!
Table List :: student, hobby, instructor, subject,
```

생성된 테이블에 레코드 삽입

```
Insert Record To Table
------------------------------
Please input table name to insert record : subject
Insert value for column name subject_id : 00000001
Insert value for column name credit : 3
Insert value for column name semester : 2022-01
Insert value for column name subject_name : Database System
```

삽입된 레코드 정보 조회

```
Select * From Table
------------------------------
Please input table name to search every data : subject
read starts from : 4053 to : 4096
Printing total 1 records..
Record index (0)
null bitmap : fffffff0
fixed_len_column :
00000001
3
2022-01
var_len_column :
Database System
```

```
00004f90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00004fa0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00004fb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00004fc0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00004fd0  00 00 00 00 00 FF FF FF  F0 30 30 30 30 30 30 30  .........0000000
00004fe0  31 33 32 30 32 32 2D 30  31 00 00 00 1C 00 00 00  132022-01.......
00004ff0  0F 44 61 74 61 62 61 73  65 20 53 79 73 74 65 6D  .Database System
```

5-6. 테이블의 컬럼 목록 출력 테스트

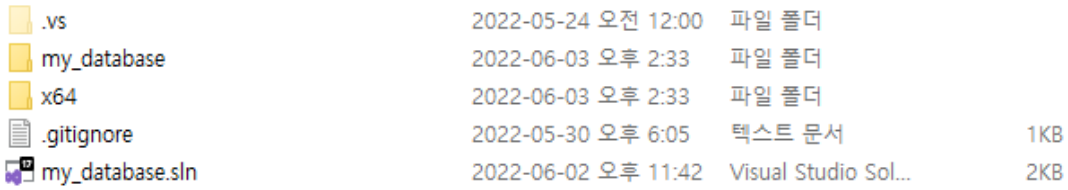
```
Select Column List of Table
-----------------------------
Please input table name to search column list : subject
column 0 : subject_id (PK)
column 1 : credit
column 2 : semester
column 3 : subject_name
```

실제 메타 데이터에 저장되어 있는 컬럼 명 목록

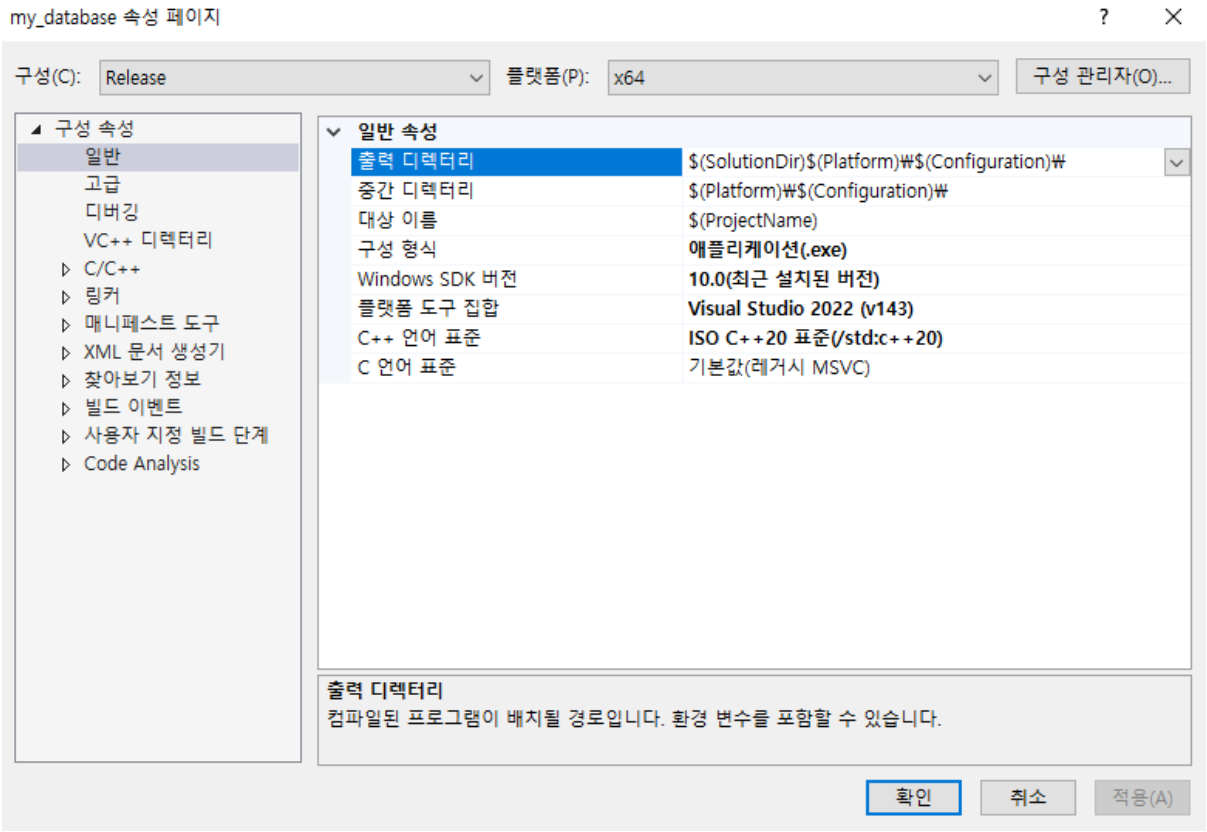

```
{
    "block_location" :
    [
        {
            "end_loc" : 20480,
            "file_name" : "data.db",
            "start_loc" : 16384
        }
    ],
    "fixed_column_cnt" : 3,
    "fixed_column_length" : [ 8, 1, 7 ],
    "pk_column_idx" : 0,
    "table_column_list" : [ "subject_id", "credit", "semester", "subject_name" ],
    "table_name" : "subject",
    "variable_column_cnt" : 1
}
```

6. 실행 파일 생성 방법 설명

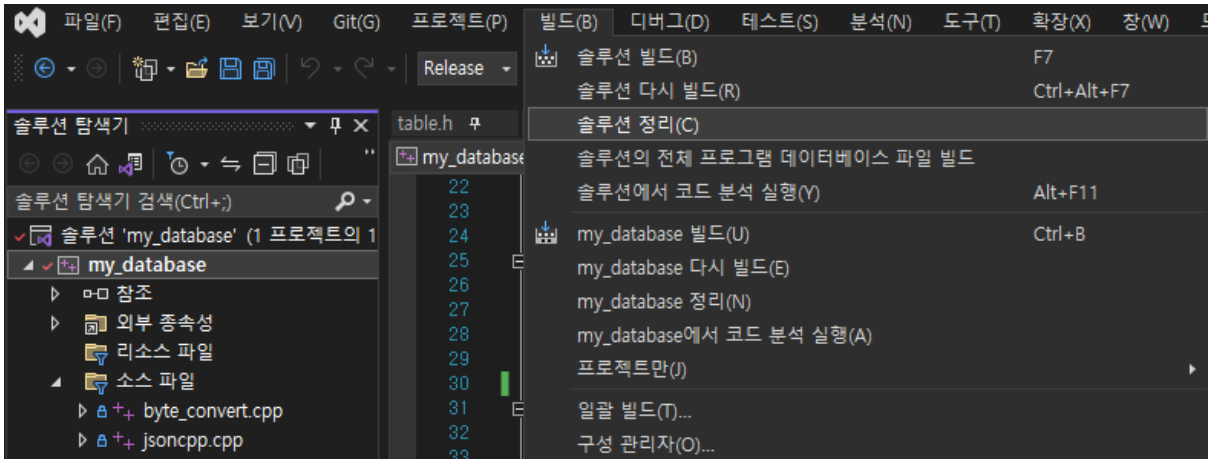
Windows 10 Home (Build) 및 Visual Studio 2022 Community 환경에서 실행 파일을 생성하는 방법은 아래와 같다.



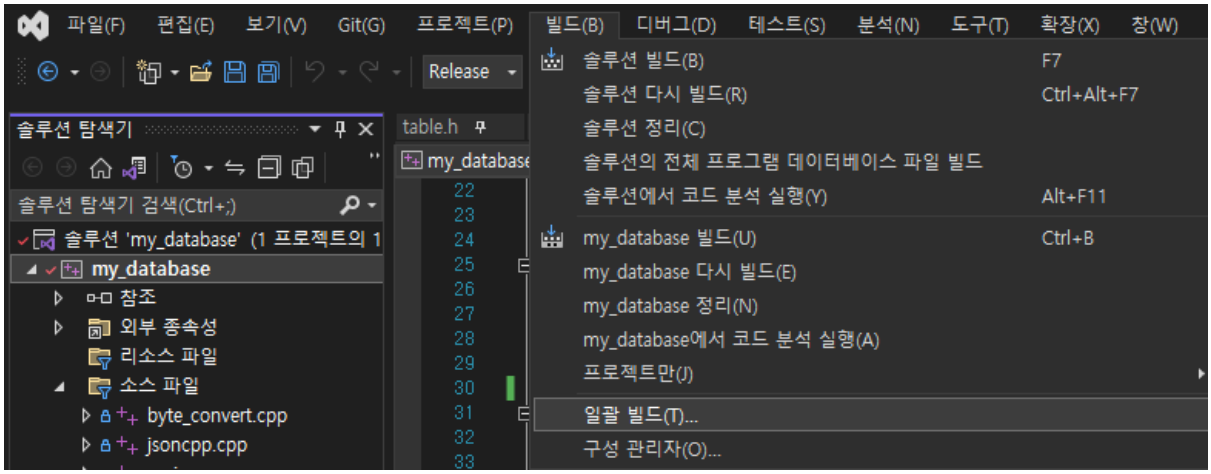
먼저 Visual Studio 2022 버전에서 src/my_database/ 폴더 안 my_database.sln 솔루션 파일을 연다.



그 후 빌드에 앞서 프로젝트 속성을 상기와 사진과 일치시켰는지 확인한다. 구성 탭이 'Release'로 되어 있는지, 'C++ 언어 표준'을 ISO C++20 표준으로 설정함에 유의한다.



이후 상단 '빌드' 탭에서 솔루션 정리를 클릭한다.



솔루션 정리 이후 '빌드' 탭의 일괄 빌드를 클릭하여 빌드 창을 연다.

다음과 같이 '구성' 탭은 Release, '플랫폼' 탭은 x64로 설정한 이후 빌드를 눌러 빌드한다.

빌드가 완료되면 /x64/release 폴더 안의 my_database.exe 실행 파일이 생성된다.

exe 실행 파일 실행 전 반드시 최상위 폴더의 meta_data.json 파일과 data.db 파일을 같은 폴더 (ex. /x64/release/) 내에 위치 시켜야 정상 작동함에 유의해야 한다.