DB PROJECT 2

20181650 안도현

0.환경 명세

OS: WINDOWS 10 64bit

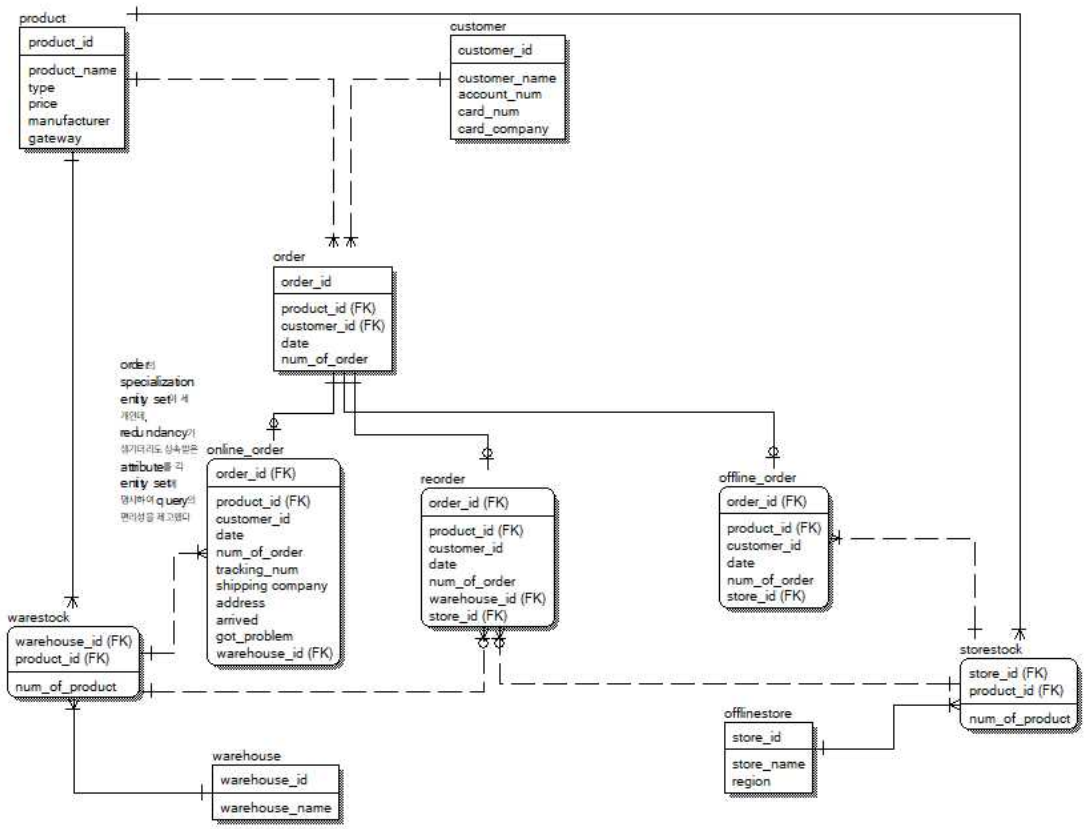
CPU: AMD Ryzen 7 5700U

GPU: AMD Radeon(TM) Graphics

Memory: 16.0 GB

IDE: Visual Studio 2022 & Mysql ODBC Driver

1. BCNF Decomposition



이는 지난 **프로젝트 1**에서 만들어둔 logical schema diagram이다. 먼저 DB의 functional dependency를 전부 파악해보면,

product_id -> product name,type,price,manufacturer,gateway

customer_id -> customer_name,account_num,card_num,card_company

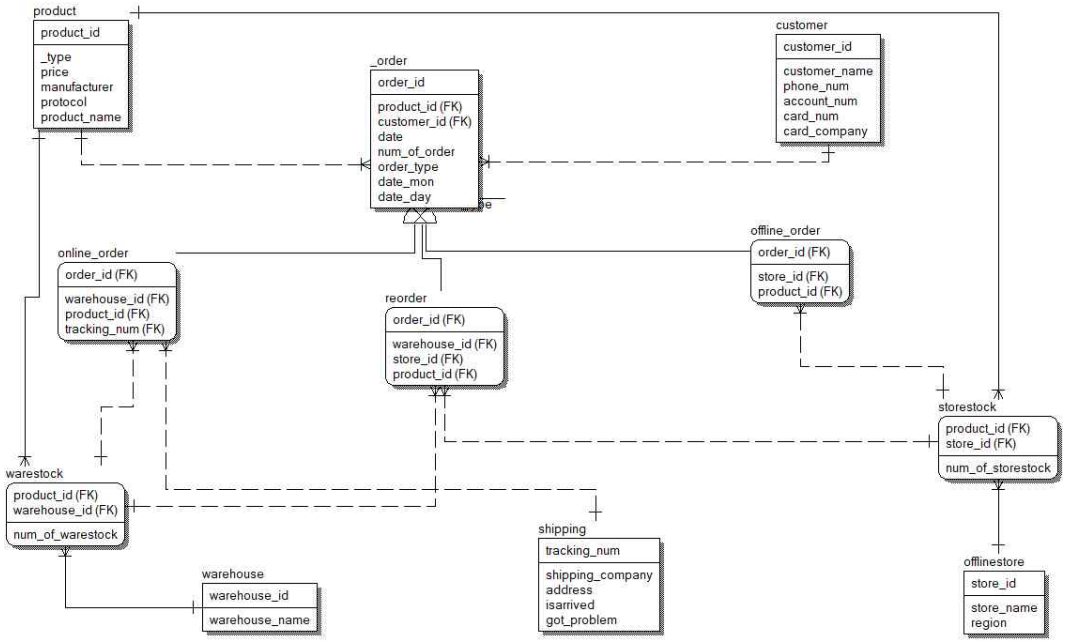
order_id -> product_id,customer_id,date,num_of_order,tracking_num,shipping company,address,arrived,got_problem,warehouse_id,store_id,order_type

warehouse_id -> warehouse_name

warehouse_id,product_id -> num_of_product

store_id -> store_name,region

store_id,product_id -> num_of_product이다.
근데 여기서 추가적으로 online_order에서 tracking_num->shipping company,address,arrived,got_problem,warehouse_id를 정의하는 편이 직관적으로 옳은 듯하여 그 부분을 추가할텐데, 그렇게 하면 BCNF가 아니게 되고 다른 수정할 부분도 있기 때문에 그것들을 수정하여 새로 다이어그램을 작성할 것이다. 그 결과로 만들어진 다이어그램은 아래와 같다.

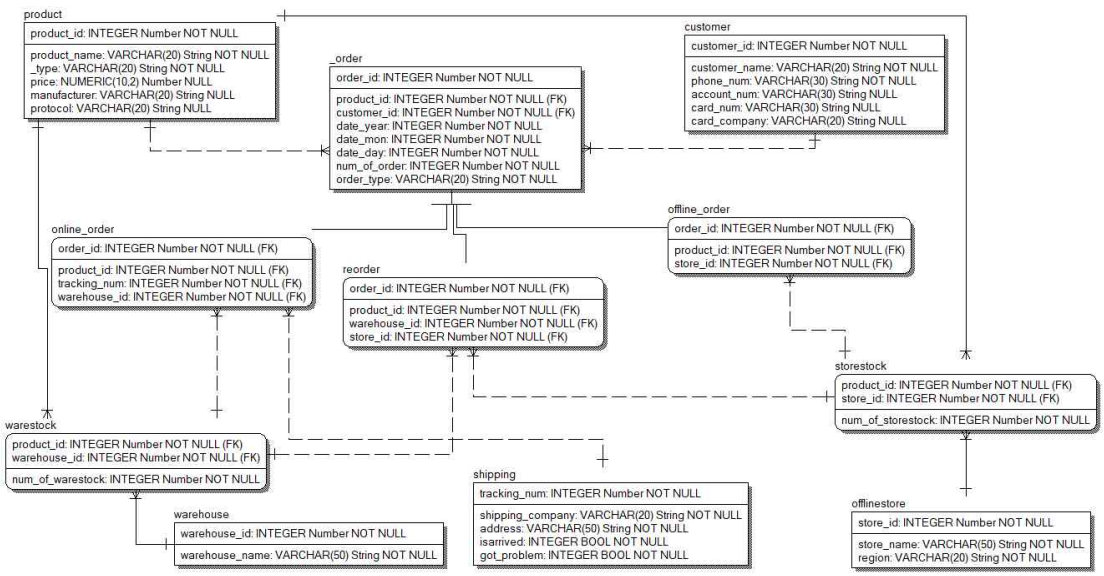


이 다이어그램에 대한 설명은 뒤에 physical diagram에 대한 설명과 함께 하도록 하고, 우선 릴레이션별로 하나씩 테스트 해보기 앞서 테스트는 강의자료에 있는 방식대로 F+가 아닌 F에 대해서 진행하되, 그 릴레이션의 모든 속성 부분집합에 대해 그 속성 집합의 closure가 릴레이션 내의 다른 속성을 포함하지 않거나 모든 속성을 포함하는지를 검사할 것이다. 즉, 속성의 closure이 trivial하거나 그 속성 집합이 릴레이션의 superkey면 BCNF를 violate하지 않는 것이다.

product	product_id를 포함한 모든 집합은 superkey이며, 포함하지 않은 모든 집합은 해당하는 functional dependency(이하 f.d)의 결정자(left side에 포함된 속성)가 아니므로 trivial한 closure이다. 따라서 BCNF를 만족한다.
customer	customer_id를 포함한 모든 집합은 superkey이며, 포함하지 않은 모든 집합은 해당하는 f.d의 결정자가 아니므로 trivial하다. 따라서 BCNF를 만족한다.
order	order_id를 포함한 모든 집합은 superkey이며, 포함하지 않은 모든 집합은 해당하는 f.d의 결정자가 아니므로 trivial하다. 물론 product_id와 customer_id는 특정 f.d의 좌측에 존재하지만, order 릴레이션의 속성들은 결정짓지 않는다. 따라서 BCNF를 만족한다.
online_order	order_id를 포함한 모든 집합은 superkey이며, 포함하지 않은 모든 집합

	은 해당하는 f.d의 결정자가 아니므로 trivial하다. 모든 릴레이션의 속성들이 foreign key이므로 헷갈릴 수 있지만, 역시 online_order 내의 속성들을 결정하지는 않는다. 따라서 BCNF를 만족한다. 또 order 릴레이션에 이렇게 몇개로 나뉜 정보들을 모두 저장할 수도 있었겠지만, 그렇게 할 시 주문의 타입과 order_id가 각 주문 종류별로 값을 가지는 속성들을 결정짓는 종속성이 나타나므로 이렇게 별개의 릴레이션으로 나누었다.
shipping	tracking_num이 다른 모든 속성들을 결정짓는 f.d를 만족하도록 깔끔하게 분리한 릴레이션으로 당연히 BCNF를 만족한다.
offline_order	order_id를 포함한 모든 집합은 superkey이며, 포함하지 않은 모든 집합은 해당하는 f.d의 결정자가 아니므로 trivial하다. 역시 store_id,product_id 등은 상점 정보와 물건 정보에 대해서만 결정하기에 BCNF를 만족한다.
reorder	order_id를 포함한 모든 집합은 superkey이며, 포함하지 않은 모든 집합은 해당하는 f.d의 결정자가 아니므로 trivial하다.
warehouse	warehouse_id 홀로나 warehouse_name과 함께 구성된 집합이나 둘 다 superkey이고, warehouse_name은 그 자체로 trivial한 종속성만 가지므로 BCNF이다.
warestock	warehouse_id와 product_id를 모두 갖고 있는 집합은 superkey, 아닌 경우 trivial 한 종속성이므로 BCNF를 만족한다.
offlinestore	store_id가 primary key이자 유일한 f.d의 결정자이고 다른 속성들은 그렇지 않으므로 바로 BCNF임이 보인다.
storestock	store_id와 product_id를 모두 갖고 있는 집합이 superkey이고 아닌 경우 trivial한 종속성이므로 BCNF이다.

2. Physical Schema diagram



우선 프로젝트 1에서 작성했던 다이어그램과 달라진 속성 구성과 릴레이션에 대해 간단히 살펴보면, order 릴레이션에 주문의 종류를 저장하는 order_type 속성을 도입했고, 추후 쿼리의 용이성을 위해 composite attribute인 date를 세 개의 속성으로 나누었다. 또 기존엔 세

가지의 order 종류와 그 부모 릴레이션인 order의 관계를 일대다 관계로 표현했었으나 그것은 표현히 적절하지 못하다고 생각해 exclusive generalization 기호를 가져와 표현했다. tracking_num이 결정짓는 다른 속성들 때문에 BCNF를 완성하기 위해 shipping 테이블을 새로 만들었다. order와 type이 SQL문에서 사용되는 예약어이기 때문에 _order,_type으로 테이블 이름을 변경했고, num_of_product로 창고와 상점에서 이름이 겹치던 속성을 다르게 바꿔주었다. 쿼리에 필요한 정보로 고객의 전화번호가 필요했기에 그 속성도 추가했다. 마지막으로 product 테이블에서 기존에 gateway라고 오명명했던 속성을 protocol로 바꾸어 더 직관적으로 어떤 값이 저장되는지 알 수 있도록 했다.

product	product_id를 primary key로 설정했고, product_name과 type은 상품 관리에 필수적인 요소라고 생각해 NOT NULL 옵션을 붙였다. 사실 NULL의 사용이 크게 보면 단점이 많기 때문에 이 다이어그램의 대부분 속성은 NOT NULL 옵션을 주었는데, price,manufacturer,gateway 옵션은 어떤 상품의 프로토타입 같은 것을 시험적으로 게시할 경우를 대비해 NULL값도 넣을 수 있도록 했다. price는 소수점이 생길 수 있는 자료형이므로 NUMERIC 타입을 채용했다. warestock과 storestock 릴레이션에 상품 정보를 넘기기 위한 식별관계(identifying relationship), order 릴레이션에 상품 정보를 넘겨 주문 정보에서 상품을 참고할 수 있도록 비식별관계(non-identifying relationship)를 설정했다. price는 0보다 크도록 하는 제약을 두어 이상한 값을 받지 않도록 했다.
customer	customer_id를 primary key로 설정했고, customer_name은 NOT NULL, 그 외 결제 정보들은 NULL이 허용되도록 설정했다. 계약된 고객과 그렇지 않은 고객, 또 카드 정보를 가진 고객과 아닌 고객으로 분류한다면 할 수 있었겠지만, 어차피 아무런 결제 정보가 없고 고객 이름과 ID만 가진 튜플을 넣을 수 있다고 한다면 NULL이 들어가야만 하므로 쿼리의 편리함을 위해 결제 정보에 해당하는 속성을 모두 통합했다. 그리고 상점의 재고가 떨어졌을 때 창고로부터 가져오는 주문인 reorder 또한 고객의 주문과 비슷한 정보, 즉 결제 정보 및 주문자 이름 등이 필요하다고 생각해 그들의 정보도 customer 테이블로 일괄 저장 및 처리한다. 카드 번호와 계좌 번호는 대소비교를 하지 않아 integer로 표시하기엔 비효율적이라고 생각해 그냥 string으로 저장했다. 이 타입을 사용함으로써 '-'같은 기호도 원하는 위치에 적절히 삽입할 수 있다. order 릴레이션과 비식별관계를 맺고 있는데, 이는 주문 정보에서 고객 정보를 올바르게 참조할 수 있게 하기 위함이다.
order	order_id를 primary key로 설정했고, 외래키 product_id와 customer_id를 사용해 두 릴레이션의 정보를 받는다. 여기서 order_id를 추가로 생성한 이유는 상품과 고객이 일치해도 여러 주문 정보가 발생할 수 있어 primary key가 너무 많은 속성으로 이루어질 수 있기 때문이었다. 또 상술했듯 date는 원래 하나로 사용하려 했으나, 년도 또는 월별로 데이터를 수집해야 하는 쿼리 요구사항이 있기에 composite value의 각 원소들을 모두 속성으로 변환했다. order_type은 사실 굳이 string 도메인을 할 필요없이 0,1,2의 수로 각 주문의 종류를 표현하는 것이 더 효율

	적인 저장 방식이겠지만, 여기서는 readability를 위해 그냥 online,offline,reorder의 string type을 갖는다. 이 테이블로부터 online_order,offline_order,reorder이 파생되어 그 속성들을 상속받고, 이는 주문 종류에 따라 관계없는 속성이 전부 NULL로 채워지는 결과를 예방하는 효과가 있다. 역시 year,month,day의 최소와 최댓값을 적절히 제약조건으로 넣어주고, num_of_order은 0보다 큰 조건을 두어 이상한 값의 삽입을 막았다.
online_order	order_id를 primary key로 두어 그 존재를 order 테이블에 의존하는 테이블이다. 프로젝트 1에서는 쿼리 단계의 편리를 위해 이 주문 테이블들에 모두 order 테이블의 속성들을 중복해서 저장했는데, 여기서는 저장이 더 간결하고 중복 저장을 회피하는 방향으로 선회하여 자식 테이블들에서 order 테이블의 기본 속성들을 모두 제거했다. 각 속성들은 모두 foreign key인데, 이들은 결국 모두 다른 테이블의 primary key로부터 정보를 받아오는 테이블이라는 뜻이다. 위에서도 그랬듯 product_id를 통해 order을 거쳐 product 테이블의 정보를 받는다. 또 후술할 shipping 테이블로부터 tracking_num을 이용해 정보를 받아오고 warehouse_id을 이용해 창고의 재고 정보를 받아오는데 이 둘은 모두 비식별 관계이다. online order는 배송에 문제가 생기거나 제때 도착하지 않는 경우가 있다. 이때 다시 배송을 보내게 되면, 다른 내용은 동일한 채 tracking_num만 업데이트된다.
offline_order	online_order와 동일한 형식이다. 다만 여기서는 창고 정보 대신 store_id를 통해 상점 재고 정보를 받는다는 것 다르다. 추가적으로 이 id들은 길지 않은 수로 저장할 예정이라 모두 integer type으로 되어 있고, 당연히 주문에는 대응되는 상품이나 재고 정보가 연결되어야 하기 때문에 NULL을 허용하지 않는다.
reorder	역시 위 두 주문 종류 테이블과 동일한 형식이다. order_id로 튜플을 유일하게 결정하고, product_id를 통해 상품 정보를 받아오고, 또 상점의 재고를 채우고 창고의 재고를 그만큼 차감하는 형식이기 때문에 두 재고 정보를 받는 warehouse_id와 store_id를 테이블에 저장한다. 이 재고 정보 또한 비식별 관계로 각 재고 테이블과 연결되어 재주문에 필요한 정보들을 전부 저장할 수 있다.
offlinestore	store_id를 primary key로 설정해 각 상점을 unique하게 결정할 수 있다. 이 상점 정보는 우선 상점의 이름과 지역을 저장하는데, 여기에는 적절한 데이터 관리를 위해 필수적으로 이름과 지역을 기입해 NULL이 될 수 없도록 했다. 상점의 이름의 길이가 길어질 수 있으므로 길이는 최대 50자가 될 수 있도록 타입을 결정했다. storestock 테이블과 식별 관계로 이어져 상점의 재고를 저장한 storestock이 상점의 정보를 받아갈 수 있도록 했다.
warehouse	warehouse_id를 primary key로 설정해 창고의 이름과 더불어 유일하게 튜플이 결정되도록 했다. 디테일한 부분은 offlinestore와 동일하다. warestock과 식별 관계로 이어져 창고의 재고 정보를 가지고 있는

	warestock 테이블이 창고의 정보를 warehouse_id를 이용해 받아갈 수 있도록 했다.
warestock	다른 테이블들과는 다르게 primary key를 두 개의 속성으로 설정했는데, 창고의 정보를 결정하는 warehouse_id와 해당하는 상품의 정보를 저장하는 product_id가 그것이다. 두 속성의 묶음으로 warestock의 튜플을 유일하게 결정하고, 이 속성들에 유일하게 일반 속성인 num_of_warestock는 그 상품의 재고가 몇개인지를 저장할 정수 변수로서 선언했다. product id는 product 테이블과 식별 관계로 이어져 상품 정보를 받아오고, 창고의 재고 정보가 필요한 주문인 온라인 주문과 상점의 재주문에 재고 정보를 제공한다. 재고는 음수가 될 수 없도록 제약 조건을 두어 오류를 예방한다.
storestock	역시 primary key는 두 개이다. 스키마를 재설계할 때 BCNF 완성을 위해 offlinestore을 따로 분리했기 때문에 product_id와 더불어 상점 정보를 유일하게 결정하는 store_id를 외래키로 사용한다. 상점 재고 정보가 필요한 주문인 오프라인 주문과 상점의 재주문과 연결되어 그 정보를 제공한다. 재고는 음수가 될 수 없도록 해 쓰레기 값으로부터 오류가 발생하지 않도록 한다.
shipping	프로젝트 1에서는 존재하지 않았던 테이블이다. tracking_num으로부터 다른 속성들을 결정짓는, 즉 직관적으로 볼 때 송장번호로 그 수신 주소나 도착 여부 등을 확인하여 key로 작용하는 것같은 f.d가 존재하기에 BCNF Decomposition을 거친 결과 얻은 테이블이다. 그래서 tracking_num이 primary key로 설정되었고, 그로부터 NULL값을 허용하지 않을 운송회사와 50자 길이의 string type 주소가 우선 존재한다. 또 도착했는지 여부와 문제가 생겼는지를 나타낼 속성인 isarrived와 got_problem는 도메인에 BOOL이라는 것을 따로 정의해 표시했는데, 실상은 integer 타입으로 정의했다. mysql 타입에서 bit이 존재하여 이를 사용하면 효율적이라고 생각했지만, 더 안정적이고 제약없이 사용할 수 있는 integer을 사용하기로 결정했다. 이 두 변수는 그래서 1 또는 0으로만 T/F를 표시하게 된다. online_order 테이블이 이 테이블로부터 tracking_num을 외래키로 가지면서 운송 정보를 조회할 수 있도록 했다.

3. TYPE QUERY&Code Implementation

우선 cpp 코드는 크게 보면 세 파트로 나뉘어져 있다. Create,Insert sql문이 적힌 txt파일을 읽어와 그 쿼리들을 요청함으로써 테이블을 생성하고 튜플들을 삽입하는 구간, 그리고 TYPE QUERY를 진행하기 위해 돌아가는 main loop, 마지막으로 Delete,Drop이 sql문이 적힌 txt 파일을 읽어 데이터베이스를 clean up하는 구간이다. 여기서는 Type Query를 위한 코드가 메인이기 때문에 그를 중점으로 살펴보도록 하겠다.

먼저 선언한 함수부터 살펴보면,

```
int putSelectMessage() {
    int input;
    cout << "\n---------- SELECT QUERY TYPES ----------\n\n";
    cout << "\t1. TYPE 1\n" << "\t2. TYPE 2\n" << "\t3. TYPE 3\n" << "\t4. TYPE 4\n" << "\t5. TYPE 5\n"
        << "\t6. TYPE 6\n" << "\t7. TYPE 7\n" << "\t0. QUIT\n\n";
    cout << "USER INPUT: ";
    cin >> input;
    return input;
}
```

putSelectMessage함수는 한눈에 보이듯 selection message를 출력하고 stdin에서 user의 input을 받아오는 함수이다. 메인함수를 조금이라도 더 간단히 하고자 따로 작성했다.

```
void query_print(MYSQL_RES* sql_result, int n=MAXVAL) {
    MYSQL_ROW sql_row;
    int num_fields;
    num_fields = mysql_num_fields(sql_result);  //필드(속성)의 개수
    while ((sql_row = mysql_fetch_row(sql_result)) != NULL && n > 0)
    {
        n--;
        for (int i = 0; i < num_fields; i++) {
            if (sql_row[i])
                cout << sql_row[i] << " | ";
            else
                cout << "NULL" << " | ";    //속성에 NULL이 들어갈 경우 출력을 따로 하여 오류를 예방
        }
        cout << '\n';
    }
    mysql_free_result(sql_result);
}
```

query_print 함수는 쿼리를 진행할 때마다 sql_result로부터 row를 반복해서 fetch해 출력하는 과정을 메인 함수에서 최대한 작성하지 않도록 따로 작성한 함수이다. 함수의 두 번째 인자는 테이블의 상위 몇 개의 튜플만 뽑아올 때 그 뽑아올 개수를 인지하기 위해 받아오는 인자이다. 일반적으로 모든 튜플을 출력한다고 하면 디폴트 인자인 MAXVAL(=1E9) 를 받아오기 때문에 while문에 아무런 영향을 미치지 않지만, 유의미한 n이 인자로 들어와 while을 돌며 1씩 감소하는 n이 모든 튜플을 출력하기 전에 0이 되면 출력을 끝내고 리턴한다. 이렇게 작성함으로써 범용성을 갖춘 함수를 만들었다.

메인 함수에 진입하면 일반적으로 MYSQL과 연결하는 과정을 거치는데, 이 부분은 템플릿으로 제공된 부분이라 설명을 생략한다. 이후 CREATE,INSERT를 진행하는 코드는 아래와 같다.

```
string buf; //sql 처리를 위한 버퍼
int state,num_fields;   //state=쿼리 요청 함수의 리턴값, num_fields=필드의 개수 저장
ifstream fin("20181650_create_insert.txt"); //Create와 Insert가 쓰여 있는 입력파일 open
if (fin.fail()) {
    cerr << "create&insert txt file open error!\n";
    return 1;
}
while (!fin.eof()) {
    getline(fin, buf,';');
    cout << buf;
    if (buf.empty())
        continue;
    state = mysql_query(connection, buf.c_str());
    if (state)
        printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
    buf.clear();
}
fin.close();
```

물론, 여기서 create_insert를 delete_drop으로 바꾸면 메인의 마지막에 존재하는 delete_drop 루틴과 정확히 동일한 코드이기에 같이 설명하겠다. 그저 파일을 입력받아 출력 없는 쿼리를 진행하는 것이 끝이므로 그렇다. 먼저 버퍼로 기능하는 buf를 먼저 저장한 뒤 쿼리의 결과를 리턴받을 state, 테이블의 속성의 개수를 저장할 num_fields를 선언한다. 그리고 입력 파일 스트림을 열어준 뒤 파일이 끝날 때까지 while을 돌며 입력을 받는다. 여기서 character단위로 입력을 받으면 비효율적이라고 생각해 ';'를 구분자로 한 getline을 사용하여 ;로 구분된 sql문을 하나씩 받아 요청할 수 있도록 했다. 특히 이 CRUD파일에 적힌 create와 insert는 어차피 유의미한 latency가 존재하기 때문에 프로그램의 시작과 종료 시 sql문을 그대로 한번씩 echo해서 어떤 명령이 이루어지고 있는지를 확인할 수 있도록 했다. buf가 개행 문자로만 이루어져 있으면 쿼리를 요청해도 오류만 뜨기 때문에 그때는 continue를 수행하도록 했다.

```
//query routine starts
while (1) {
    int input = putSelectMessage(); //selection message 반복 출력
    if (input == 0) {   //0 input: break-> delete/drop 진행
        cout << "---- User Requested QUIT! ----\n\n";
        break;
    }
```

그러면 이제 본격적으로 TYPE QUERY를 살펴보자. 메인에서 크게 while(1) 루프를 돌며 탈출 조건을(0 input from stdin) 주지 않으면 계속 selection message를 출력하며 반복하도록 했다. 0이 입력되면 이 큰 while문을 break하는데, 이 뒤에는 바로 delete_drop을 진행하는 코드가 존재하기 때문에 그를 수행한 뒤 프로그램이 종료되게 된다.

```
else if (input == 1) {  //1 input: tracking_num을 입력받아 그로부터 연결된 주문 정보로와 연결해 고객의 연락처를 받음.
    cout << "---- TYPE 1 ----\n\n";
    string x;
    cout << "Input tracking_num: ";
    cin >> x;
    //코드 전체에서 가장 복잡한 join. join이 무려 네번이나 이루어지지만 redundancy를 최소화한 디자인의 한계일 것이다.
    //필요한 정보인 고객 id와 이름, 연락처 속성만 표시해 가시성 제고
    buf = "select customer_id,customer_name,phone_num from ((shipping natural join online_order)inner join _orde
    state = mysql_query(connection, buf.c_str());
    if (state) {
        cerr << "TYPE 1 Error!\n";
        printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
        continue;
    }
    cout << "customer_id | customer_name | contact\n";
    cout << "-----------------------------------\n";
    sql_result = mysql_store_result(connection);
    query_print(sql_result);
```

이는 TYPE 1의 쿼리이다. stdin으로부터 tracking_num을 string 타입으로 입력받고, sql문을 저장할 buf에는

```
select customer_id,customer_name,phone_num
from ((shipping natural join online_order)inner join
_order using (order_id)) inner join customer using
(customer_id)
where tracking_num = x
```

가 저장되는데, shipping,onliner_order,_order,customer이 join된 테이블로부터 tracking_num을 조회해 customer 정보를 받는다. 연락처가 필요하므로 고객 ID, 이름, 연락처만 표시해 가시성을 높였다. 얕은 지식으로는 4번의 join이 그렇게 이상적인 연산이 아닌 것처럼 보이는데, redundancy를 최소화하여 db 자체의 신뢰성을 높이기 위해 선택한 BCNF 디자인에서 어쩔 수 없이 발생하는 한계가 아닌가 생각한다.

또 앞으로도 위와 동일한 과정의 쿼리가 이루어지는데, mysql_query에 sql문이 저장된 buf를 전달하고, state를 통해 오류를 검출하고, 적절한 형식의 속성 이름을 출력해 준 뒤 sql_result에 mysql_store_result함수로 결과를 저장해, 이를 위에서 정의한 query_print함수에 넘김으로써 stdout에 출력한다.

```
if (subinput == 1) {
    cout << "-----TYPE 1-1 -----\n\n";
    buf = "select tracking_num from shipping";  //shipping 테이블의 튜플의 개수를 계산하기 위한 쿼리
    state = mysql_query(connection, buf.c_str());
    sql_result = mysql_store_result(connection);
    sql_row = mysql_fetch_row(sql_result);
    int maxship = 100000 + mysql_num_rows(sql_result);  //base tracking_num=100000 + row개수 = maxship
    maxship++;  //새 운송 정보에 부여할 tracking_num.
    _itoa(maxship, temp, 10);
    string tmp = temp;

    buf = "select * from shipping where tracking_num=\'" + x + "\'";    //1에서 입력받은 넘버로 운송정보 조회
    state = mysql_query(connection, buf.c_str());
    sql_result = mysql_store_result(connection);
    sql_row = mysql_fetch_row(sql_result);
    cout << "tracking_num | company | address | isarrived | got_problem\n";
    cout << "-------------------------------------------------------------\n";
    cout << sql_row[0] << " | " << sql_row[1] << " | " << sql_row[2] << " | " << sql_row[3] << " | " << sql
    mysql_free_result(sql_result);
```

```
//새롭게 갱신한 운송정보 삽입
buf = "insert into shipping values(" + tmp + ",\'" + sql_row[1] + "\',\'" + sql_row[
cout << "\nNow processing " << buf << '\n';
state = mysql_query(connection, buf.c_str());
if (state) {
    cerr << "TYPE 1-1 Error!\n";
    printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
}

//주문 정보에서 기존의 문제가 생긴 tracking_num 자리에 새로운 넘버를 갱신한다.
buf = "update online_order set tracking_num= " + tmp + " where tracking_num=" + x;
cout << "\nNow processing " << buf << '\n';
state = mysql_query(connection, buf.c_str());
if (state) {
    cerr << "TYPE 1-1 Error!\n";
    printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
}
```

위는 1-1 QUERY의 과정이다. 먼저 shipping table의 튜플의 개수를 판단하기 위해 간단한 쿼리를 한번 날린 뒤 mysql_num_rows함수를 통해 그 값을 얻는다. 새로 shipping 테이블에 튜플을 추가하기 위해 이 과정이 필요한데, 그 최댓값에 1을 더한 값을 새 튜플의 tracking_num으로 주기 때문에 그렇다. 그리고 TYPE1에서 입력받은 tracking_num에 대응되는 shipping table의 튜플을 유일하게 찾아내 먼저 출력한다. 이 튜플이 가진 정보를 복사해 새로운 튜플에 집어넣어 줘야 되기 때문에 이전 쿼리에서 받은 sql_row 정보를 사용하는데, 그 쿼리가 아래와 같다.

```
insert into shipping
values( tmp,sql_row[1],sql_row[2],0,0);";
```

tmp는 maxship에 1을 더해 새롭게 뽑아낸 tracking_num이고, sql_row[1]과 sql_row[2]는 이전 쿼리에서 얻은 company와 address정보, 그리고 0 두개는 isarrived와 gotproblem을 우선 둘 다 0으로 초기화하기에 넣은 값이다.

또 문제가 생긴 shipping 정보를 가지고 있는 online_order 테이블의 특정 튜플의 tracking_num을 새롭게 추가한 tracking_num으로 갱신하여 다시 시작한 운송 정보를 참조할 수 있도록 업데이트 쿼리를 진행했다.

위 두 쿼리는 결과를 출력하지 않는 쿼리이기 때문에 사용자 입장에서 뭐가 어떻게 되고 있는지 궁금할 수 있다. 따라서 위 insert와 update문은 stdout에 한번 echo하도록 했다.

```
else if (input == 2) {  //2 input: 작년에 가격 기준 가장 많이 구매한 고객 정보를 받음.
    cout << "---- TYPE 2 ----\n\n";
    //with절을 사용해 작년 구매 정보를 먼저 받고, 그로부터 max값을 얻도록 sql문 작성
    buf = "with max_buy_price as (select customer_id, customer_name, date_year, sum(price * num_of_orc
    buf = buf + "where date_year = " + pastyear + " group by customer_id) select customer_id, customer
    state = mysql_query(connection, buf.c_str());
    if (state) {
        cerr << "TYPE 2 Error!\n";
        printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
        continue;
    }
    cout << "customer_id | customer_name | pricesum\n";
    cout << "-------------------------------------\n";
    sql_result = mysql_store_result(connection);
    sql_row = mysql_fetch_row(sql_result);
    cout << sql_row[0] << " | " << sql_row[1] << " | " << sql_row[2];
    string bestmanid = sql_row[0];  //subquery에서 사용할 vip고객의 id정보 별도 저장
    mysql_free_result(sql_result);
```

TYPE2 QUERY이다. 복잡한 쿼리를 한번에 끝내기 위해 with절을 사용했다. 쿼리문은 아래와 같다.

```
with max_buy_price as
(select customer_id, customer_name, date_year, sum(price * num_of_order) as pricesum
from(_order natural join customer) inner join product using(product_id)
where date_year = "pastyear"
group by customer_id)

select customer_id, customer_name, pricesum
from max_buy_price
where pricesum = (select max(pricesum) from max_buy_price);
```

max_buy_price는 우선 주문과 고객, 상품 테이블을 join 시킨 테이블에서 customer_id로 묶은 뒤, 작년에 구매 기록이 있는 고객의 정보와 그 고객의 구매 액수를 출력하는 쿼리 문이다. 그래서 여기서 pricesum의 max값과 같은 pricesum을 가지고 있는 튜플, 즉 구매 액수가 작년에 가장 큰 튜플을 뽑아내 그 고객 정보와 구매 액수를 출력하도록 한 것이다. 여기서 query_print 함수를 또 쓰지 않은 이유는 이후 서브쿼리에서 이 고객의 ID를 사용할 것이기 때문에 sql_row[0]을 따로 저장할 필요가 있기 때문이었다.

2-1 QUERY는 2번에서 얻은 최우수 고객이 당해 가장 많이 구매한 상품의 정보를 출력하는 쿼리이다.

```
with max_buy_unit as
(select product_id, product_name, _type, price, manufacturer, date_year,
sum(num_of_order) as sum_order
from(_order natural join customer) inner join product using(product_id)";
where customer_id = bestmanid and date_year = pastyear
group by product_id)

select *
from max_buy_unit
where sum_order = (select max(sum_order) from max_buy_unit); ";
```

크게 보면 2번의 쿼리와 유사한 부분이 많다. 다만 이 쿼리의 with 절에서는 product_id로 묶고, 또 customer_id가 2번 쿼리에서 얻은 고객의 id이다. 이렇게 얻은 max_buy_unit 테이블에서 sum_order, 즉 주문 개수가 최댓값을 갖는 상품의 정보를 출력한다. 여기서는 해당 상품의 id만 달랑 내놓지 않고 거의 모든 속성을 출력해 해당 상품에 대해 디테일한 정보를 얻을 수 있도록 했다.

3번과 4번의 TYPE Query는 형식이 거의 동일하므로 같이 설명하겠다.

```
else if (input == 3) {
//주문 정보, 고객 정보, 상품 정보를 join하여 작년에 주문된 기록이 있는 주문 정보를 view에 저장한다.
buf = "create view pastyearsell as select * from _order natural join customer natural join product where date_yea
    state = mysql_query(connection, buf.c_str());
    if (state) {
        cerr << "View generating error before TYPE3!\n";
        printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
        continue;
    }
    cout << "---- TYPE 3 ----\n\n";
    //팔린 금액을 sum한 뒤 product_id별로 묶어 해당하는 속성들을 출력한다.
    buf = "select product_id, product_name, _type, price, manufacturer,sum(num_of_order*price) as sold_total from
    state = mysql_query(connection, buf.c_str());
    if (state) {
        cerr << "TYPE 3 Error!\n";
        printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
        continue;
    }
    cout << "product_id | product_name | type | price | manufacturer | sold_total\n";
    cout << "-------------------------------------------------------------------------\n";
    sql_result = mysql_store_result(connection);
    query_print(sql_result);
```

3번과 4번은 그 처리를 하기 앞서 동일한 view를 생성했다. view 생성 쿼리는 아래와 같다.

```
create view pastyearsell as
select *
from _order natural join customer natural join product
where date_year = pastyear
```

주문 정보와 고객, 상품 정보를 join한 테이블에서 그 주문 날짜가 작년인 모든 속성을 출력하는 것이다. 기본적으로 3번과 4번은 작년에 이루어진 주문으로부터 어떤 정보를 얻는 것들이기 때문에 여러번 같은 쿼리를 진행하기보다 뷰를 통해 수행하는 것이 코드 길이 면에서 훨씬 좋을 것이라고 판단했다. 물론 이렇게 뷰를 이용하는 블록의 마지막에는

DROP VIEW [view name]

의 쿼리를 통해 삭제를 해주었다.

아래는 3번 쿼리의 sql문이다. 작년에 판매된 상품에 대한 속성들과, 그 상품이 총 판매된 금액을 sold_total이라는 이름을 붙여 표시했는데, 상품을 기준으로 묶고 금액 기준으로 내림차순 정렬했다. 여기서 굳이 판매 총금액과 그 내림차순 정렬을 한 이유는 아래에서 살펴보겠지만 subquery 2가지가 모두 이 정보를 이용하기 때문이다.

```
select product_id, product_name, _type, price, manufacturer,sum(num_of_order*price) as
sold_total
from pastyearsell
group by product_id
order by sold_total desc;
```

```
cin >> subinput;
if (subinput == 1) {    //k를 입력받아 3의 결과에서 top k 튜플을 뽑아 출력한다.
    cout << "-----TYPE 3-1 -----\n\n";
    cout << " Which K ? : ";
    cin >> subinput;
    buf = "select product_id, product_name, _type, price, manufacturer,sum(num_of_order*price) as sold_
    state = mysql_query(connection, buf.c_str());
    sql_result = mysql_store_result(connection);
    cout << "product_id | product_name | type | price | manufacturer | sold_total\n";
    cout << "-----------------------------------------------------------------------\n";
    query_print(sql_result, subinput);
}
else if (subinput == 2) {   //3의 결과에서 top 10%의 튜플을 출력하기 위해 row개수/10을 연산 후 그만큼 출력한
    cout << "-----TYPE 3-2 -----\n\n";
    buf = "select product_id, product_name, _type, price, manufacturer,sum(num_of_order*price) as sold_
    state = mysql_query(connection, buf.c_str());
    sql_result = mysql_store_result(connection);
    int topten = (int)mysql_num_rows(sql_result) / 10;
    cout << "product_id | product_name | type | price | manufacturer | sold_total\n";
    cout << "-----------------------------------------------------------------------\n";
    query_print(sql_result, topten);
}
```

이는 3번의 서브쿼리인 3-1과 3-2에 해당하는 코드이다. 3-1은 k를 stdin으로부터 입력받고, 3번과 동일한 쿼리를 진행한 뒤, query_print의 두번째 인자로 이 k값을 넘겨준다. 그럼 판매 금액 기준으로 내림차순 정렬되어있는 테이블에서 상위 튜플부터 k개만 출력한 뒤 함수를 리턴하기 때문에 원하는 결과를 얻을 수 있다. 물론 이 과정을 sql 문으로 진행하지 못할 것은 없지만, 그렇게 하면 쿼리문이 너무 복잡해지고 또 odbc 자체의 장점을 채용해 이렇게 c++ 코드 내에서 제어하는 게 더 이상적인 방향이라고 생각했다.
3-2도 마찬가지이다. 다른점은 오직 query_print의 두번째 인자 뿐인데, topten이라는 변수를 따로 선언해 sql_result의 튜플의 개수에서 10을 나누어 전체의 10% 개수를 구해 그만큼만 출력하도록 한 것이다.

그렇다면 4번은 3번의 과정과 뭐가 다를까? 기존에 코드에서 sold_total로 표현했던 sum(num_of_order*price)를 sold_total_unit으로 바꾸어 sum(num_of_order)로 바꾸기만 하면 모든 것이 동일하다. 요는 판매 금액 기준으로 표시했던 것들을 모두 판매 개수로만 따지면 되는 것이다.

5번과 6번은 단순히 쿼리 문을 입력해 요청하고 결과를 query_print 함수로 출력하는 것이 전부이므로 sql문 자체만 보도록 하겠다. 5번의 buf에 채워지는 내용은 아래와 같다.

```
select product_id,product_name,_type,manufacturer,region,sum(num_of_storestock) as sum
_of_storestock
from offlinestore natural join storestock natural join product
where region = 'california'
group by product_id
having sum_of_storestock = 0;";
```

먼저 상점과 상점재고, 상품 정보의 테이블을 join한 뒤 california 지역에 있는 튜플만 뽑아내고, product_id 기준으로 묶어 상품의 재고가 0인 튜플이 있다면, 그 튜플의 상품 정보들과 지역(california), 그리고 sum_or_storestock(0)을 출력한다. 여기서 상품 정보들 외에 굳이 지역과 재고 개수가 고정인데도 써놓은 이유는 출력된 데이터가 무엇인지 더 잘 보이게 하

기 위함이다.

6번 쿼리는 아래와 같다.

```
select *
from shipping
where got_problem=1;
```

매우 단순하다. join 없이 이미 운송 정보를 가지고 있는 shipping table에서 got_problem이 set된 테이블만 찾으면 되기 때문이다.

+추가)
만약에 6번 쿼리에서 조금 더 깊은 정보를 위해 그렇게 문제가 발생한 order 정보를 찾는 쿼리라면 아래와 같이 복잡해진다.

```
select order_id,tracking_num,customer_id,shipping_company,address,isarrived
from (online_order natural join shipping) inner join _order using (order_id)
where got_problem=1;
```

online_order와 shipping, 그리고 _order을 join한 테이블에서 문제가 생긴(got_problem이 set된) 튜플을 찾아 해당하는 속성들을 끄집어낸다. 다만 여기서 1-1번 쿼리와 연결이 되는데, 1-1번에서 문제가 생긴 tracking_num에 대해 새로 넘버를 부여해 튜플을 삽입하고, order에는 tracking_num을 바꿔버리기 때문에 위와 같이 쿼리를 하면 문제가 생겼었던 주문 정보는 출력하지 않는 특징이 생긴다. 따라서 용도에 따라 적절한 sql문을 취사선택하면 될 것이다.

```
else if (input == 7) {
    //지난 달에 팔린 주문 정보와 그 주문에 관계된 고객 정보를 뷰에 저장
    buf = "create view pastmonthsell as select * from _order natural join customer natural join pro
    state = mysql_query(connection, buf.c_str());
    if (state) {
        cerr << "View generating error before TYPE4!\n";
        printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
        continue;
    }
    cout << "---- TYPE 7 ----\n\n";
    //bill은 contracted customer(account num is not null)에 대해서만 적용하므로, 해당 조건을 걸어준다.
    buf = "select customer_id,customer_name,order_id,product_name,price,num_of_order,price*num_of_o
    state = mysql_query(connection, buf.c_str());
    if (state) {
        cerr << "TYPE 7 Error!\n";
        printf("%d ERROR : %s\n", mysql_errno(&conn), mysql_error(&conn));
        continue;
    }
    sql_result = mysql_store_result(connection);
    int n_contracted = (int)mysql_num_rows(sql_result); //bill을 만들 튜플의 개수
```

7번 쿼리를 위한 코드이다. 여기서도 뷰를 먼저 생성하는데, 3,4번에서와 다른 점은 그냥 조회하는 일자가 작년이 아니라 지난 달인 것만 다르다. 여기서 지난 달에 대한 정보는 이 코드의 전역변수로서 선언해 달마다 바꿔야 하는 것으로 제약했고, 현재 지난 달은 2022-5로 저장되어 있다. 7번의 sql문은 아래와 같다.

```
select customer_id,customer_name,order_id,product_name,price,num_of_order,price*num_of_
order as total_purchase
from pastmonthsell
where account_num is not null
```

지난 달의 구매 정보를 담고 있는 pastmonthsell view로부터 고객과 그 주문 정보 등을 표시하는 쿼리인데, account_num이 null이 아닌 튜플만 표시한다. 즉, contracted customer이어서 계좌 정보가 저장되어 있는 고객에 대해서만 달마다 bill을 생성해 전송하기 때문에 그에 맞추어 조건을 걸어준 것이다. 그런 제약을 없애고자 하면 그냥 이 sql문에서 where절만 지우면 된다. 그렇게 하면 아래 코드와 같이,

```
while ((sql_row = mysql_fetch_row(sql_result)) != NULL) {
    cout << "-----------------------------------------\n";
    cout << "|\tcustomer_id: " << sql_row[0] << '\n';
    cout << "|\tcustomer_name: " << sql_row[1] << '\n';
    cout << "|\torder_id: " << sql_row[2] << '\n';
    cout << "|\tproduct_name: " << sql_row[3] << '\n';
    cout << "|\tprice: " << sql_row[4] << '\n';
    cout << "|\tnum_of_order: " << sql_row[5] << '\n';
    cout << "|\ttotal_price: " << sql_row[6] << '\n';
    cout << "-----------------------------------------\n\n";
}
mysql_free_result(sql_result);

buf = "DROP VIEW pastmonthsell";    //뷰 제거
state = mysql_query(connection, buf.c_str());
continue;
```

결과로 받은 모든 튜플에 대해 적절한 형식에 맞추어 bill을 만들어 stdout으로 출력한다. 물론 block에 끝에도 view를 제거하는 쿼리를 수행한다.

마지막으로 위에서 설명한 것처럼 이 while문을 탈출하면 delete_drop을 파일 입력으로부터 수행하고 프로그램을 종료하게 된다.