Project 1-3: Implementing DML

본 프로젝트는 SQL 문을 파싱하고, 스키마를 저장하고 관리하고, 직접 데이터를 저장·삭제·검색·수정할 수 있는 기능을 구현하는 것이 목표이다.

핵심 모듈

- **`grammar.lark`**: EBNF (Extended Backus-Naur Form) 형식으로 SQL 문법을 정의한다. Lark API 를 사용하면 이 파일에서 정의된 문법을 기준으로 SQL 쿼리를 파싱하고 그 결과를 AST(Abstract Syntax Tree)로 변환할 수 있다.
- **`sql_transformer.py`**: Lark 의 `Transformer` 클래스를 상속한 `SQLTransformer` 클래스에서 파서에서 생성한 AST 를 처리한다. 쿼리에서 제공하는 테이블, 레코드, select 하는 컬럼들 등 데이터를 처리하여 반환한다.
- **`db_model.py`**: 스키마와 레코드의 자료구조 (각 `Table, Record` 클래스) 정의되어 있고, BerkeleyDB 의 DB 객체를 조작하는 wrapper 인 `DB` 클래스가 있다. 스키마의 메타데이터를 담는 DB 는 DB 클래스를 상속한 `MetaDB` 로 정의된다.
- **`dbms.py`**: CREATE TABLE, DROP TABLE, EXPLAIN/DESCRIBE/DESC, SHOW TABLES, INSERT, DELETE, SELECT 구문을 처리하는 DBMS 클래스가 있다.
- **`messages.py`**: `DBMS` 클래스가 SQL 명령을 성공적으로 수행했음을 알리는 로그와 에러 메시지를 출력하는 예외 클래스들을 정의한다.
- **`utils.py`**: `Unknown` 변수, `Unknown` 을 포함한 SQL 의 논리연산, 파싱된 비교/null 연산자에 대응되는 함수 매핑을 정의한다. 또한 date 데이터 타입을 포함하여 데이터 타입의 유효성을 검증하는 함수들을 정의한다.
- **`run.py`**: 쿼리 시퀀스를 여러 개의 구문으로 나누고, 각 쿼리별로 동작을 수행한다. Lark 라이브러리로부터 `Lark` 클래스를 불러와 `grammar.lark` 파일에서 정의된 문법을 기준으로 하는 파서를 생성하고, `SQLTransformer` 를 불러와 파서가 생성한 AST 를 해석하여 필요한 데이터를 추출한다. SQL 구문에 대응되는 처리 함수를 `DBMS` 인스턴스에서 호출하고, 결과물 또는 에러 메시지를 출력한다.

구현 내용 및 알고리즘

- **`grammar.lark`**: INSERT 구문이 null 값도 받을 수 있도록 null 데이터 타입을 추가한다.
- **`sql_transformer.py`**: `Transformer` 클래스는 AST 를 bottom-up 순서로 순회하므로, 리프 노드를 발견한 경우에는 이를 파싱한 결과물을 반환하고, 상위 노드, 특히 쿼리를 직접적으로 식별하는 노드에서는 하위 노드들로부터 수합한 데이터를 `table, record, tables, select_columns, where` 정보로 분류하여 iterable 에 저장한다. 그리고 가장 상위 노드에 다다를 때 이 데이터들을 반환한다. 테이블 이름과 컬럼 이름은 모두 소문자로 변환하고, int 값을 받을 때는 파이썬의 int, null 값을 받을 때는 파이썬의 None 으로 type casting 한다. Where 절에서는 `predicate, Boolean_factor, Boolean_term` 을 파싱할 때 연산자와 피연산자를 구분하여 dictionary 에 저장함으로써 최종적으로 where 절의 내용이 nested dictionary 로 담기게 처리한다.
- **`db_model.py`**
    - 스키마의 메타데이터를 별도의 DB 파일에 저장, 관리하고 (Metadata schema), 하나의 DB 파일에 하나의 테이블의 레코드를 모두 담는 (one DB-one schema) 방식을 사용한다. 전자인 `MetaDB` 클래스에서는 key 를 테이블 이름, value 를 Table 인스턴스로 BerkeleyDB 의 `DB` 인스턴스에 저장하며, 후자인 `DB` 클래스에서는 key 를 레코드의 primary key 또는 무작위로 생성한 UUID (primary key 가 없을 시), value 를 `Record` 인스턴스로 저장한다. 하나의 DB 파일 안에 테이블 키와 레코드 키가 혼재할 경우 탐색 비용의 비효율성을 줄이기 위함이다. `Table` 과 `Record` 인스턴스는 pickle 라이브러리를 이용하여 serialize 한다.
    - `Table` 과 `Record` 클래스 모두 어떤 테이블이나 레코드로부터 참조되고 어떤 컬럼이나 레코드 값을 참조하는지를 관리한다. 이를 통해 DROP TABLE, INSERT, DELETE 시 integrity constraint 를 빠르게 확인할 수 있다.
- **`dbms.py`**

  - 한 `DBMS` 인스턴스에서는 `MetaDB` 인스턴스를 지속적으로 관리하면서 테이블 메타데이터를 필요 시 가져온다. 각 테이블에 있는 레코드에 접근해야 할 때는 DB 인스턴스를 생성하고, 파일을 열고, BerkeleyDB 의 `DBCursor` 객체를 이용하여 레코드를 순회한다.
  - INSERT 시, 만약 컬럼이 foreign key 에 해당하는데 입력하고자 하는 값이 참조되는 테이블의 primary key 값에 없다면 InsertReferentialIntegrityError 를 발생시킨다. 참조되는 테이블의 primary key 가 composite key 인 경우에도 작동한다. 또한 DB 의 key 가 primary value 인 것이 default 이기 때문에 BerkeleyDB 에서 제공하는 함수로도 InsertDuplicatePrimaryKeyError 를 확인하는 것이 편리하다.
  - DELETE 시, 삭제하고자 하는 레코드의 값 중 참조되는 것이 하나라도 있다면 DeleteReferentialIntegrityPassed 의 값을 증가시킨다. 삭제하려는 레코드가 참조하는 값들이 있다면 해당 값들이 본래 있는 레코드에서 참조되고 있다는 표시를 삭제한다. 이를 통해 INSERT, DELETE 구문 모두에서 레코드 간의 referential integrity 를 제어할 수 있다.
  - SELECT 시, from 절에 있는 테이블들의 컬럼들 중 공통되는 것들을 추출하고, 각 테이블에 있는 레코드를 모두 가져오는데 이때 공통 컬럼의 값들은 대응되는 컬럼 앞에 테이블 이름을 붙인다. 공통 컬럼들을 처리한 모든 레코드들을 파이썬 itertools 라이브러리의 product 함수에 전달하여 cartesian product 을 만든다. 이후 where 절이 있다면 where 조건들을 만족시키는 레코드들만 필터링, select 되는 컬럼들이 명시되어 있다면 레코드들의 project 하면서 컬럼 이름을 주어진 이름과 순서에 맞도록 구성한다. 최종적으로 남겨진 레코드들을 출력할 때 파이썬 None 으로 처리한 값들은 null 로 변환한다.
  - Where 절을 평가하는 함수는 재귀적으로 구현되었다. `SQLTransformer` 로부터 받은 where 절의 nested dictionary 로써 가능하며, 연산자가 비교/null 이면 해당 연산을 수행한 뒤 True/False/Unknown 값을 반환하고, 이후 operator precedence 에 입각한 조건문들에서 연산자가 not, and, 또는 or 일 경우에는 하위 조건들에 대해 함수를 재귀적으로 호출한다. 가장 상위에 호출될 때는 전체 where 절의 nested dictionary, 테이블 리스트, 그리고 레코드 데이터가 전달되며, 하위의 Boolean test 에서 레코드 데이터를 평가한다.
- **`utils.py`**: 연산자 각각에 해당하는 함수를 파이썬 operators 라이브러리로부터 불러와, where 절을 평가할 때 피연산자의 개수와 구애받지 않고 동작할 수 있도록 한다.
- **`run.py`**
  - **`main` 함수**:
    1. `grammar.lark` 파일에서 정의된 문법을 기준으로 `Lark` 로부터 파서를 생성한다.
    2. `SQLTransformer` 의 인스턴스를 생성한다.
    3. `exit` 변수가 참이 아닐 때,
       i. 쿼리(시퀀스)를 입력받고 `process_query_sequence` 함수에서 이를 쿼리의 목록으로 처리한다.
       ii. (목록 내 각) 쿼리에 대해 `parse` 함수를 실행하여 쿼리로부터 필요한 데이터를 추출한다.
       iii. 메시지가 "exit"일 경우에는 `exit` 변수를 거짓으로 바꾸고 루프를 종료하고, 그렇지 않을 경우에는 구문이 지시하는 대로 스키마와 데이터를 관리한다.
       iv. `parse` 함수를 실행하는 과정에서 틀린 구문이 발견될 경우 에러 메시지를 출력하고, 쿼리들이 남아있을 경우에는 더이상 처리하지 않는다.
       v. A 부터 반복한다.

<u>느낀 점 및 기타사항</u>

스키마를 어떻게 하면 효율적이면서 기능에 충실하게 구현할 수 있을지 많은 고민을 했고, design pattern 에 대한 필요를 느꼈다. 많은 테스트 케이스를 작성하며 코드의 정밀성을 강화할 수 있었다.

Metadata Schema, One DB-One schema 방식을 채택하고, key-value pair 로 하나하나의 테이블 메타데이터 또는 레코드를 관리하였지만, One DB-Multi schema 방식을 차용했을 경우 파일을 관리하기 더 편할 수 있겠다는 생각이 들었다.

Referential integrity 를 관리하기 위해 각 테이블 메타데이터와 레코드에 표식을 달아두었는데, BerkeleyDB 단에서 제공하는 기능을 충실히 활용할 방법을 모색해본다면 덜 번거로울 수 있겠다고 생각했다.