Proj 1-2 (Implementing DDL)

컴퓨터공학부 2014-10469 한인성
2019-04-21

핵심 모듈과 알고리즘에 대한 설명

저번 프로젝트(1-1)에서 만든 파서에 Oracle Berkeley DB API 추가하여 실제 데이터베이스를 구현했다. 이번 프로젝트에서는 여러 query중 DDL에 해당하는 query을 실행하면 실제로 *persistent*한 table 생성, 삭제, 출력이 가능하도록 구현했다. 프로젝트를 진행하면서 가장 핵심적인 모듈과 원리에 대해서 설명하고자 한다.

- **(Structure of table) table에 대한 정보를 어떻게 저장할 것인가?**
  Project 1-3과 함께 고려하면서, table에 대한 정보 **(schema)** 를 담는 database와 실제 record **(instance)** 를 담는 database를 분리하기로 결정했다. 왜냐하면 하나의 database에 schema와 instance를 모두 담게 되면 **insert**와 **delete**를 할 때 마다 테이블 전체를 봐야하기 때문에 *scaleable*하지 않기 때문이다.
  따라서 이번 프로젝트에서 구현하는 database는 `value` 로 schema만을 가지며 그것을 찾을 `key` 로 **table의 이름** 을 정했다. schema는 `class Table` 의 객체로 구현했고 이 객체는 `table_name` , `attributes` , `primary_keys` , `referencing` , `referenced` 로 구성되어 있는데 여기서 특별한 점은 **attributes(columns)** 와 **primary key** 를 분리해서 저장했다는 것이다. `primary_keys` 에 들어있는 attribute들은 모두 `attributes` 에도 물론 들어있어야한다. 단, `attributes` 는 `Attribute` 객체를 담고 있어 좀 더 attribute에 대해 자세한 정보를 담고 있지만 `primary_keys` 는 단순히 String의 `ArrayList` 를 담고있다. `referencing` 은 `class Reference` 객체를 담고있는 `ArrayList` 이고 **foreign key**와 foreign key가 참조하는 table 및 대응되는 attribute를 담고있다. 마지막으로 `referenced` 는 해당 `Table` 객체를 foreign key로 참조하는 table 의 `table_name` 을 담고 있다. 위와 같이 `Table` 을 구성한 이유는 에러처리에서 좀 더 용이하게 하기 위해서이다.
- **어떻게 parsing 하면서 query한 정보를 가져올 수 있을까? & 에러처리**
  parsing하면서 *Consume*된 Token이나 BNF production의 return값을 가지고 주어진 query를 실제 database에 적용하고 에러에 대한 처리를 해주어야한다. 한편, database에 적용하는 것과 에러처리는 반드시 **Syntax Error**를 체크한 뒤에 한번에 진행되어야 한다. 왜냐하면 그 전까지는 query가 전부 *Consume*된 것이 아니기 때문이다. 만약 그렇게 하지 않는다면 `create table` query를 받고 parsing을 진행하면서 table을 database에 단계적으로 추가하면 database는 수정되었지만 마지막에 query에 문법 오류가 있어서 **Syntax Error**가 출력될 수 있기 때문이다. 이 문제는 `query()` function을 시작하기 전에 `queryConfig` 라는 특별한 객체를 call by reference로 `query()` 함수에 넘겨주고 이후 `query()` 가 부르는 함수

들에 대해서도 비슷한 과정을 거침으로써 해결했다. `queryConfig` 는 database 적용에 필요한 정보들을 담고 있는 클래스로 `queryList()` 에서 정의되어 그 이하의 함수에 대해 *recursive*하게 넘겨준다. 각 `query()` 가 끝난 뒤에 즉 **Syntax**를 확인한 뒤에 `queryConfig` 에는 생성된(만약 `create table` query가 불렸다면) table의 schema(`Table`)와 요청하는 `table_name` (`desc` 나 `drop table` query의 경우), 그리고 `primary_def_count` (primary 정의부 개수)가 남는다. 그 뒤에는 `query()` 종류에 따라서 `queryConfig` 에 저장된 객체를 사용하면 된다.

구현한 내용에 대한 간략한 설명

Serialize

serialize 는 객체를 저장할 때 해당 database에서 넣고 뺄 수 있도록 적당한 Byte 배열로 변환하는 것을 의미한다. 여기서 `value` 저장되는 `Table` 객체도 database에 넣고 뺄 수 있도록 serialize가 필요한데,
`Table` 객체는 복잡한 Object이기 때문에 단순히 *primitive data type*처럼 `getBytes()` 함수를 이용할 수가 없다. 이런 경우에는 우선 `Table` 의 **member** 의 class 와 `Table` class가 `Serializable` interface를 **implement** 해야한다. 그 이후 아래와 같이 `myClassDb` 에 클래스 정보를 저장하고 나면

```
dbConfig.setSortedDuplicates(false);
myClassDb = myDbEnvironment.openDatabase(null, "classDb", dbConfig);
classCatalog = new StoredClassCatalog(myClassDb);
dataBinding = new SerialBinding(classCatalog, Table.class);
```

`entryToObject()` 함수를 이용하여 `Table` ↔ `DatabaseEntry` 간 변환을 할 수 있다.

```
table = (Table) dataBinding.entryToObject(foundData);

dataBinding.objectToEntry(table, data);
```

가정한 것들

1. **정의의 순서와 무관하게 처리된다.**
   예를들어 아래와 같이 primary key와 해당 attribute가 반대로 선언 되어도 올바른 query로 본다.

```
create table abc(
    primary key(gg);
    gg int not null
    );
```

2. **spec에 명시된 에러가 아닐 경우** foreign key는 무조건 reference하는 table의 모든 primary key를 참조한다.
   예를들어 다음과 같이 참조 table의 primary key를 부분적으로 참조하는 경우는 발생하지 않는다.

```
create table A (
    a1 int, a2 int, a3 int,
    primary key(a1, a2, a3)
);
create table B (
    b1 int, b2 int,
    foreign key(b1, b2) references table1(a1, a2)
```

4. **데이터베이스는 query 에 의하지 않고서는 삭제되거나 추가되지 않는다.**

5. **입력으로 들어올 수 있는 문자는 다음과 같다.**

- Alphabet(a~Z)
- number(0~9)
- special characters on the keyboard
- new line, carrige return, space

컴파일과-실행-방법">컴파일과 실행 방법

proj1-1과 마찬가지로 컴파일 방법은 동일하다.

compile

eclipse에 **javacc plug-in**을 설치하게 되면 매번 저장할 때마다 자동으로 'jj'파일을 컴파일 해준다. 만약 자동으로 컴파일 되지 않는다고 해도 'jj'파일을 우클릭한 뒤 **compile with javaCC**를 선택하면 수동으로 컴파일 할 수 있다.

실행방법

실행도 마찬가지로 _eclipse_에서 **run**버튼을 눌러서 프로젝트를 바로 실행할 수 있다.
혹은, 실행파일로 만들어서 실행하고 싶다면 해당 프로젝트를 우클릭 하고 **Export -> Runnable JAR File**을 선택한 뒤 **launch configuration**에 해당 프로젝트를 넣고 **finish**를 누르게 되면 Jar파일을 생성한다.
생성된 Jar 파일을 실행하기 위해서는 **terminal**에서 해당 jar 파일 위치로 간 뒤에 아래 shell 명령어를 실행시키면 된다.

```
java -jar <file name>.jar
```

단, 해당 `jar` 파일이 위치한 directory에 database를 위한 `db` 폴더가 있어야한다.

프로젝트를 하면서 느낀 점

relational table을 어떻게 key-value DB로 표현할 것인지 고민을 많이 할 수 있었다. DB에 대한 operation 종류와 사용빈도에 따라서 어떤 자료구조를 사용하여 어떻게 design할 것인지가 달라질 수 있다는 것을 깨달았다. 상당히 코드가 길고 복잡해져서 디버깅에 어려움이 있을까 걱정 했지만 control flow가 단순해서 , 다행히도 큰 오류 없이 마칠 수 있었던 것 같다. 다만, 아쉬운 점은 구현을 중시한 나머지 naming이나 코드 손질에 미비했던 것 같다.