Logical Querying of Relational Databases

Luminita Pistol¹, Radu Bucea-Manea-Tonis²

¹ Spiru Haret University
² Hyperion University

Abstract. This paper aims to demonstrate the usefulness of formal logic and Lambda Calculus in database programming. After a short introduction in propositional and first order logic, we implement dynamically a small database and translate some SQL queries in filtered java 8 streams, enhanced with Tuples facilities from jOOλ library.

Keywords: logic query, propositional logic, predicate, relational database

JEL Codes: M15

1. Introduction

A database is a set of basic axioms corresponding to base relations and tuples plus deductive axioms or inference rules. Tuples are for the relationships what are nouns for sentences, each denote a true particular sentence [Date, 2005].

A logical query is the action of evaluating a Boolean expression concerning tuples and relations. Boolean operators in propositional logic are:

<table>
<thead>
<tr>
<th>Operator name and meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation (non)</td>
<td>¬φ</td>
</tr>
<tr>
<td>conjunction (and)</td>
<td>(φ &amp; ψ)</td>
</tr>
<tr>
<td>disjunction (or)</td>
<td>(φ</td>
</tr>
<tr>
<td>implication (if ..., then ...)</td>
<td>(φ -&gt; ψ)</td>
</tr>
<tr>
<td>equivalence (if and only if...)</td>
<td>(φ &lt;-&gt; ψ)</td>
</tr>
</tbody>
</table>
A basic axiom is equivalent to a tuple of a database or a predicate. The predicate value is a function of truth that has a set of parameters. It should not be assigned a value to a database in order to determine the database predicate to take the truth value FALSE [Date, 2005]

Table 2: Basic axiom table

<table>
<thead>
<tr>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caninae</td>
<td>Canis</td>
</tr>
<tr>
<td>Canis</td>
<td>Canis lupus</td>
</tr>
</tbody>
</table>

Corresponding to the example above, we can construct an open formula with two occurrences of the variable x:

Grandparent(x) <- Parent (x) & (Child(x) <-> Parent(y))

By placing an existential quantifier ∃ before x (“for some x”) and an universal quantifier ∀ before y (“for all y”), we can bind these variables, as may be seen bellow [Bird, 2009]:

∃x. ∀y. Grandparent(x) <- Parent (x) & (Child(x) <-> Parent(y))

1.1. Advantages of logical querying:

- Uniform representation of operations and dependency constraints;
- Improved semantics of the original data model;
- Improve SQL facilities making possible to negate a where clause if we keep in mind the formal logic rules [StackOverflow, 2016]:

A & B & (D | E) ↔ □ (A & B & (D | E)) ↔ □A | □B | (□D & □E)

2. Case study

Suppose that in our database the following scheme has been defined [Moshe, 2006]:

Student (name, dorm, major, GPA),


Professor (name, dept, salary, year hired)
Chair (dept, name)
We create a dynamic structure for this as the following:

```java
Studenti = new ArrayList<Student>();
Profesori = new ArrayList<Professor>();
Decani = new ArrayList<Chair>();
Decani.add(new Chair("Iosipescu","Math"));
Decani.add(new Chair("Radulescu","CS"));
Profesori.add(new Professor("Georgescu","CS",5000,1999));
Profesori.add(new Professor("Iosipescu","Math",3000,2004));
Profesori.add(new Professor("Radulescu","CS",7000,2000));
Profesori.add(new Professor("Marinescu","Math",6000,1998));
Studenti.add(new Student("Ionescu", "A5", "CS", 9.5));
Studenti.add(new Student("Marinescu", "A3", "Math", 9.0));
Studenti.add(new Student("Popescu", "A4", "CS", 8.5));
Studenti.add(new Student("Vasilescu", "A5", "Math", 7.5));
```

2.1. **List the name and dorm of Math students with a GPA of at least 8.0:**

```java
List<Student> result = db.Studenti.stream().filter(s -> s.major.equals("Math") && s.GPA>=8.0).collect(Collectors.toList());
```

2.2. **List the names of faculty members with a salary to 5000, who were hired after 1990:**

```java
List<Professor> result1 = db.Profesori.stream().filter(p -> p.salary<=5000 && p.year>=1990).collect(Collectors.toList());
```

2.3. **List the names of faculty whose salary is higher than their chair's salary:**

```java
db.Profesori.stream()
.sorted((p1, p2) -> Long.compare(p1.salary, p2.salary))
.flatMap(v1 -> db.Decani.stream())
.filter(v2 -> Objects.equals(v1.dept, v2.dept)) && db.Profesori.stream()
```
61. anyMatch(t -> v1.salary>t.salary && t.name.equals(v2.name)))
   .map(v2 -> tuple(v1.name, v2.name))
   .forEach(System.out::println);

2.4. List the names of faculty members whose salary is highest in their department:

   db.Profesori.stream().filter(p->db.Profesori.stream().anyMatch(t->t.salary<p.salary &&
   t.dept.equals(p.dept))).forEach(p->{System.out.println("name="+p.name);});

   We have employed the jOOλ library [GitHub, 2016], making the following mappings [Fusco, 2015]:

   INNER JOIN - flatMap() with filter()
   WHERE - filter()
   GROUP BY - collect()
   HAVING - filter()
   SELECT - map()

   The results are the following:

   name=Marinescu dorm=A3//1
   name=Georgescu
   name=Iosipescu//2
   (Marinescu, Iosipescu)//3
   name=Radulescu
   name=Marinescu//4.

3. Conclusions

   There are advantages. Evaluating expressions and functional programming has already given us the
   support for a declarative way of parsing collections of objects. Since relational databases cease way to
   noSQL ones, we have to discover a good substitute for SQL language. Beginning with Java 8 lambda
   expressions, streams and method references, we have to search no more...

4. References


