Leveraging Synergy Between Database and Programming Language Courses

Brian Howard
DePauw University

This work was supported by the 2008–11 Donald E. Town Faculty Fellowship from DePauw University.
Motivation and Overview

- Enhance DB or PL course by building on connections with the other
- Examples:
  - Syntax-Directed SQL Translation
  - Comprehension Syntax
  - Object-Relational Mapping
  - Transactional Memory
  - Document-Oriented Databases
  - MapReduce
Syntax-Directed SQL Translation

Grammar for a subset of SQL

\[
\begin{align*}
Expr & ::= \text{SELECT } \ast \text{ FWGH} \\
     & \quad | \text{SELECT (ColName} \mid \text{Agg})^+ \text{ FWGH} \\
FWGH & ::= \text{FWG} (\text{HAVING Cond})^? \\
FWG & ::= \text{FW} (\text{GROUP BY ColName}^+)^? \\
FW & ::= \text{F} (\text{WHERE Cond})^? \\
F & ::= \text{FROM TableName}^+
\end{align*}
\]
Translation Functions

\[
\begin{align*}
\mathcal{E}[\text{SELECT } * \text{ FWGH}] & = \mathcal{H}[\text{FWGH}, \emptyset] \\
\mathcal{E}[\text{SELECT (ColName } | \text{ Agg})^+ \text{ FWGH}] & = \text{project}(\mathcal{H}[\text{FWGH}, \{\text{Agg}^+\}], \{(\text{ColName } | \text{ Agg})^+\}) \\
\mathcal{H}[\text{FWG}, \text{aggs}] & = \mathcal{G}[\text{FWG}, \text{aggs}] \\
\mathcal{H}[\text{FWG HAVING Cond}, \text{aggs}] & = \text{select}(\mathcal{G}[\text{FWG}, \text{aggs}], \text{Cond}) \\
\mathcal{G}[\text{FW}, \emptyset] & = \mathcal{W}[\text{FW}] \\
\mathcal{G}[\text{FW}, \text{aggs}] & = \text{groupby}(\mathcal{W}[\text{FW}], \emptyset, \text{aggs}) \\
\mathcal{G}[\text{FW GROUP BY ColName}^+, \text{aggs}] & = \text{groupby}(\mathcal{W}[\text{FW}], \{\text{ColName}^+\}, \text{aggs}) \\
\mathcal{W}[\text{FW}] & = \mathcal{F}[\text{FW}] \\
\mathcal{W}[\text{FW WHERE Cond}] & = \text{select}(\mathcal{F}[\text{FW}], \text{Cond}) \\
\mathcal{F}[\text{FROM TableName}] & = \text{TableName} \\
\mathcal{F}[\text{FROM TableName}^+, \text{TableName}] & = \text{product}(\mathcal{F}[\text{FROM TableName}^+], \text{TableName})
\end{align*}
\]
Example Translation

What is the difference between a HAVING and a WHERE condition when there is no GROUP BY?

\[ E[\text{SELECT } \text{Min(Year)} \text{ as } Y \text{ FROM Student HAVING } Cond] \]
\[ = \text{project}(H[\text{FROM Student HAVING } Cond, \{\text{Min(Year)} \text{ as } Y\}], \{Y\}) \]
\[ = \text{project}(\text{select}(G[\text{FROM Student, } \{\text{Min(Year)} \text{ as } Y\}], Cond), \{Y\}) \]
\[ = \text{project}(\text{select}(\text{groupby}(V[\text{FROM Student}], \emptyset, \{\text{Min(Year)} \text{ as } Y\}), Cond), \{Y\}) \]
\[ = \text{project}(\text{select}(\text{groupby}(F[\text{FROM Student}], \emptyset, \{\text{Min(Year)} \text{ as } Y\}), Cond), \{Y\}) \]
\[ = \text{project}(\text{select}(\text{groupby}(\text{Student}, \emptyset, \{\text{Min(Year)} \text{ as } Y\}), Cond), \{Y\}) \]

versus

\[ E[\text{SELECT } \text{Min(Year)} \text{ as } Y \text{ FROM Student WHERE } Cond] \]
\[ = \text{project}(H[\text{FROM Student WHERE } Cond, \{\text{Min(Year)} \text{ as } Y\}], \{Y\}) \]
\[ = \text{project}(\text{select}(G[\text{FROM Student WHERE } Cond, \{\text{Min(Year)} \text{ as } Y\}], \{Y\}) \]
\[ = \text{project}(\text{groupby}(V[\text{FROM Student WHERE } Cond], \emptyset, \{\text{Min(Year)} \text{ as } Y\}), \{Y\}) \]
\[ = \text{project}(\text{groupby}(\text{select}(F[\text{FROM Student}], Cond, \emptyset, \{\text{Min(Year)} \text{ as } Y\}), \{Y\}) \]
\[ = \text{project}(\text{groupby}(\text{select}(\text{Student, } Cond), \emptyset, \{\text{Min(Year)} \text{ as } Y\}), \{Y\}) \]
Comprehension Syntax

Generalized `for` loop, based on set builder notation

**Scala Example**

```scala
val mentorPairs = for {
  mentor <- students
  other <- students
  if mentor.year < other.year &&
    mentor.major == other.major
} yield (mentor, other)
```

This is equivalent to

```scala
val mentors = students.flatMap(mentor =>
  students.withFilter(other =>
    mentor.year < other.year &&
    mentor.major == other.major
  ).map(other =>
    (mentor, other)
  )
)
```
C# LINQ Equivalent

```csharp
var mentors =
    from mentor in students
    from other in students
    where mentor.year < other.year
        && mentor.major == other.major
    select new {a = mentor, b = other};
```

SQL Equivalent

```sql
SELECT mentor.ID as a, other.ID as b
FROM Student mentor, Student other
WHERE mentor.Year < other.Year
    AND mentor.Major = other.Major;
```
Object-Relational Mapping

Java Database Connectivity (JDBC)

List mentors = new ArrayList();
Statement statement =
connection.createStatement();
String query =
    "SELECT mentor.ID as a, other.ID as b " +
    "FROM Student mentor, Student other " +
    "WHERE mentor.Year < other.Year " +
    " AND mentor.Major = other.Major;";

ResultSet results =
statement.executeQuery(query);
while (results.next()) {
    String mentorID = results.getString("a");
    String otherID = results.getString("b");
    mentors.add(new MIDPair(mentorID, otherID));
}
results.close();
@Entity
@Table(name="Student")
public class Student {
    @Id @Column(name="ID")
    private String id; // Primary key

    @Column(name="Year")
    private int year;

    @ManyToOne @JoinColumn(name="Major")
    private Department major; // Foreign key

    // usual constructors, accessors, etc. go here
}

Java Persistence API (JPA)
Java Persistence Query Language (JPQL)

```java
List mentors = new ArrayList();
String queryString =
    "select mentor, other " +
    "from Student mentor, Student other " +
    "where mentor.year < other.year " +
    " and mentor.major = other.major";

Query query =
    entityMgr.createQuery(queryString);
for (Object result : query.getResultList()) {
    Object[] pair = (Object[]) result;
    Student mentor = (Student) pair[0];
    Student other = (Student) pair[1];
    mentors.add(new MPair(mentor, other));
}
```
C# LINQ to Entities

```csharp
var context = ...;
var query =
    from mentor in context.students
    from other in context.students
    where mentor.year < other.year
        && mentor.major == other.major
    select new {a = mentor, b = other};
var mentors = query.ToList();
```
**Transactional Memory**

```scala
class Fork { val inUse = Ref(false) }

def meal(left: Fork, right: Fork) {
  // thinking

  atomic { implicit txn =>
    if (left.inUse() || right.inUse())
      retry // forks are not both ready, wait
    left.inUse() = true
    right.inUse() = true
  }

  // eating

  atomic { implicit txn =>
    left.inUse() = false
    right.inUse() = false
  }
}
```

Example from ScalaSTM library documentation
Document-Oriented Databases

JavaScript Object Notation (JSON)

```json
{
    "ID": "12-34567",
    "Name": "Ann O'Nemus",
    "Year": 2015,
    "Major": "Computer Science",
    "Home Address": {
        "Street": "123 Main",
        "City": "Springfield",
        "State": "AK",
        "ZIP": 98765
    },
    "Phones": [
        {"Type": "Home", "Number": "555-555-1234"},
        {"Type": "Cell", "Number": "555-555-5678"}
    ]
}
```
MapReduce

Example in MongoDB: count number of students per major/year

```javascript
var map = function() {
  emit({"Major": this.Major,
       "Year": this.Year}, 1)
}

var reduce = function(key, values) {
  var total = 0;
  for (index in values) total += values[index];
  return total;
}

db.runCommand({
  "mapreduce": "students", // source collection
  "map": map,
  "reduce": reduce,
  "out": "graduates" // output collection
})
```