10-2015

Leveraging Synergy Between Database and Programming Language Courses

Brian T. Howard
DePauw University, bhoward@depauw.edu

Leveraging Synergy Between Database and Programming Language Courses

Brian T. Howard
Department of Computer Science
DePauw University
Greencastle, Indiana 46135
bhoward@depauw.edu

ABSTRACT

Undergraduate courses in database systems and programming languages are frequently taught without much overlap. This paper argues that there is a substantial benefit to emphasizing some areas of commonality, both old and new, between the two subjects. Examples of such cross-fertilization that may be used to enhance one or both of the courses include query language design and implementation, object-relational mapping, transactional memory, and various aspects of the recent "NoSQL" movement.

Introduction

The impetus for this paper came when the faculty member who had taught the database course for years in our small department retired. No one else on the faculty had ever taken a database course, much less taught one, so the author, with a background in programming languages, volunteered to take on the task of learning the material, on the basis of a half-remembered connection between Prolog and database queries. After one time through the course, with a small class of mostly-forgiving students, the realization came that there were a number of threads in common with programming languages: the design and implementation of query languages, for example, as well as the previously mentioned connection with Prolog. A sabbatical leave and a university-sponsored faculty fellowship gave the opportunity to explore these similarities further, and particularly to catch up on recent research and industry trends relating these two areas. The rest of this paper consists of brief summaries of some overlapping topics, with pointers to relevant literature and suggestions for incorporating the material into one or both of the courses. Regrettably, space constraints limit the detail with which examples can be given here. The authorÕs experience so far in exploiting these connected topics has been that students appreciate seeing the interplay between multiple courses, and classroom discussions can go deeper when they realize they can relate the content of one course to material they may have seen elsewhere.

SQL Grammar and Translation

A straightforward application of the crossover comes when describing the language SQL [14]. While the full language is huge (and most implementations notoriously deviate from the standard in significant ways), it is feasible to work with formal descriptions of useable subsets of the language. For example, a context-free grammar with just a few rules can be given for simple `SELECT-FROM-WHERE-GROUP BY-HAVING` queries, and then a syntax-directed translation can be developed that compiles those queries into relational algebra.

In the context of a PL course, this small example serves as an application of some of the formal tools to a perhaps more practical situation than the typical "expression language." In the context of a DB course, introducing this bit of formalization provides a technique for students to check their intuitions about how to interpret queries. Indeed, the act of preparing such an example caused the author to discover two tricky points of these queries that are easy to miss in less formal presentations: how to handle a query with a `HAVING` clause but no `GROUP BY` clause, and the implications of SQL's implementation-dependent column name assignment for anonymous aggregations.

Comprehension Syntax

A fairly recent addition to the repertoire of common programming language control structures is the "list comprehension," sometimes known more generally as the sequence or monad comprehension. Based on the set builder notation from mathematics, it extends the typical lower-level "for" loop by abstracting away details about how to select and iterate over elements of a collection. For example, consider the following code in Scala [21], which returns a collection of potential mentorship pairs (students with the same major, where the first is more senior):

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

A comprehension is implemented by common collection operations such as map and filter. An advantage of the comprehension approach is that it gracefully extends to richer kinds of collections and enables appropriate optimizations. For example, applying the above comprehension to a "parallel collection" of students will automatically run across all of the available cores.

This story of the advantages of a higher-level, declarative style of programming is of course also behind the success of SQL and the relational database model. Indeed, the comprehension example above is quite similar to the following SQL:

```
SELECT m.ID as mentor, o.ID as other
FROM Student m, Student o
WHERE m.year < o.year AND m.Major = o.Major;
```

A database system is free to optimize the order in which the `Student` records are traversed while performing this query, as well as take advantage of additional information such as an index (for example, to limit the search for others to only those students in the mentor's major).

The SQL **SELECT** statement, along with the XQuery **FLWOR** expression [5], was the explicit inspiration for the comprehension syntax added to Microsoft's C# 3.0 and Visual Basic 9 [19]. Known as LINQ, for Language Integrated Query, it comprises an embedded domain-specific language for writing comprehension-like queries against databases, XML documents, and in-memory data structures. On the surface, all three forms of query look the same; behind the scenes, just as described above, the actions performed will be optimized for the particular kind of collection.

Objects and Databases

Much has been written over the past quarter-century (for example, [1, 2, 3, 4, 7, 18, 20, 23]) about problems involved in integrating database systems and programming languages. There has been a particular focus on working with object-oriented languages, partly because there seems to be significant common ground between data models and object models: consider for instance the similarities between E-R diagrams and UML class diagrams. However, early on it was recognized that there is an "impedance mismatch" [18] between general-purpose programming languages and database systems. Various characterizations of this mismatch have been given; Cook and Ibrahim [7] list the following issues:

- Clash between the common imperative programming model and the declarative nature of database queries;
- Tension between program compilation and query optimization;
- Emphasis on general data structures and algorithms versus restriction to the relational model;
- Concurrency based on cooperating threads versus competing transactions;
- Incompatible interpretations of null values; and
- Different approaches to modularity and information hiding.

Many technologies have been proposed over the years to deal with this mismatch. Three that are instructive to examine in an undergraduate DB course are Java Database Connectivity (JDBC [15]), Java Persistence API (JPA [16]), and the previously-mentioned Language Integrated Query (LINQ [17]).

Java Database Connectivity

JDBC provides a fairly simple abstraction layer over a database connection. Drivers are available for many systems to handle the lower-level details of the connection, so programs may work in a relatively database-independent manner.

Some difficulties with the JDBC approach include the use of a Java String to represent the SQL query, which implies that its syntax will only be examined at runtime. In cases where queries are built dynamically, care has to be taken to avoid SQL injection vulnerabilities. There is also duplication of information at the boundary between Java and SQL, to maintain a correspondence between variables on each side. Finally, the relatively low level of JDBC, for example, requiring an explicit loop through a result set to retrieve the queried data, invites common programming errors such as neglecting to free up resources, or writing inefficient code that does processing on the client side which might have been better performed on the server side.

Java Persistence API

The Java Persistence API attempts to deal with these issues by putting another layer of abstraction on top of JDBC. It automates some of the work of the object/relational mapping: by adding Java annotations to the class definitions of entities, the programmer enables the JPA to generate an entity manager, which is an object that mediates between in-memory entities and their corresponding storage representation in the database. It manages the underlying JDBC connection and result set objects, and it shields the programmer from directly writing SQL. Instead, queries are expressed in JPQL, which resembles SQL but deals with entity objects instead of the underlying relations. There are still some problems with this approach: queries are embedded as strings instead of richer objects amenable to compile-time checking, and down-casts are required to attach the correct types to the results of queries. More substantially, it is still quite possible to write inefficient code by using the entities in such a way that the entity manager is unable to submit sufficiently optimizable queries to the database server. However, JPA represents a significant advance towards the goal of incorporating object persistence into a general-purpose programming language.

Language Integrated Query

Embedded domain-specific languages such as LINQ improve on the situation with JPA by expressing queries directly as source language constructs. This enables substantial compile-time checking of query syntax and typing, and allows the programmer to apply the full range of abstraction mechanisms (parameterization, modularization, and access control) to writing queries, while maintaining the ability of the database server to perform appropriate optimizations.

Transactional Memory

With the increasing importance of concurrent programming as the number of cores per processor increases, and the unfortunate difficulty of writing correct concurrent programs, there have been a large number of proposals for abstractions with which to manage the complexity. One that has gained popularity in recent years is transactional memory, based on the long-standing use of transactions to manage concurrency in database systems [13, 22, 11]. The concept is

simple: regions of code containing references to shared data are marked "atomic," with the guarantee that all changes to the shared data made within such a block will happen *atomically* (that is, as if all the changes were performed simultaneously) and in *isolation* (that is, without any other process changing the shared data during execution of the block).

A number of mechanisms have been devised to implement this, both in hardware and software. A typical implementation of software transactional memory (STM) uses optimistic execution, where atomic blocks are executed on the assumption that no other block has read or written to the shared memory in a way to violate the guarantees. If a violation is detected (for example, by comparing the values at the end of the block with the initial values, just before atomically writing any changes), then the transaction is "rolled-back"—any changes are undone, and the transaction is retried. This involves some overhead, but the claim is that most of the time the optimistic execution will succeed, and the small amount of extra work done behind the scenes is worth the simplified programming model, since it is much easier to reason about the correctness of the atomic blocks than to show that equivalent code with explicit locks is correct (safe and deadlock-free). The analogy is made with automatic garbage collection [12]: a skilled programmer can probably write more efficient correct code with explicit memory management, but for most purposes it is more practical to accept a small performance hit in exchange for much easier development.

NoSQL

The term "NoSQL" has become popular in the past few years to refer to a broad collection of technologies that are alternative or complementary to SQL-based relational database systems. Some of the characteristics of NoSQL approaches include an emphasis on distributed storage and processing, greater flexibility in the format of stored values, replacement of the full relational model in favor of a simpler key/value store model, and, frequently, a relaxation of the traditional ACID properties: for example, by guaranteeing only "eventual" consistency. Examples include Google's BigTable [6], Amazon's Dynamo [10], and Apache's CouchDB [8]. This paper will not attempt to debate the merits of the NoSQL approach, but simply look at two typical aspects with connections to programming languages.

Document-Oriented Databases

While some NoSQL approaches, such as BigTable, have a traditional structure of rows and columns (albeit with greater flexibility to add columns as needed), many systems store values that are documents of "semi-structured data." Typically these documents are arranged in a self-describing format such as XML or JSON (JavaScript Object Notation). A document may consist of an arbitrary collection of fields, each of which may in turn contain a complex sub-document or list of documents. This ability to store various pieces of related information in a single document alleviates much of the mismatch between objects and relational tables (and explains the reduced need for join operations).

MapReduce

The MapReduce framework [9] was developed by Google to handle a common processing task: examine all of the documents spread across a very large distributed storage system, and collect up some kind of summary information from each. The framework handles the details of farming out the job to a large number of processors, efficiently and robustly dealing with the low-level communications and data handling. All that a user of the framework needs to provide is a pair of functions, inspired by common iteration patterns in functional languages: a "map" function is applied to each document, running in parallel as much as possible, returning ("emitting") zero or more key/value pairs; a "reduce" function is then applied to combine a group of key/value pairs into a summary value or set of values. The reduce operation is typically applied in a tree fashion: individual processing nodes may apply it locally, and then send the results back to parent nodes to be reduced further. We have already seen the map operation in the context of comprehension-based queries; it is also analogous to the `FROM` and `WHERE` clauses of SQL, while the reduce operation corresponds more to the `GROUP BY` and `HAVING` clauses. Indeed, the CouchDB system uses MapReduce as the basis for defining query views.

Conclusions

We have presented a number of examples of intersections between topics suitable for undergraduate database and programming language courses. They reflect a variety of ways in which the viewpoint of one subject may be exploited to gain additional insight into the other. The selection is not meant to be comprehensive, but it is intended to cover a range of both classical and current topics; by using this sort of crossover material in a PL or DB course, the author believes that students will both gain a deeper understanding of the topic at hand and also come to appreciate the interconnectedness of many areas of computer science, which are too often seen as isolated and independent.

Acknowledgments

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

1. REFERENCES

[1] Atkinson, M. P., Buneman, O. P., Types and persistence in database programming languages, *ACM Computing Surveys*, 19(2), 105–190, 1987.

[2] Atkinson, M. P., Morrison, R., Orthogonally persistent object systems, *VLDB Journal*, 4, 319–401, 1995.

[3] Bloom, T., Zdonik, S. B., Issues in the design of object-oriented database programming languages, *OOPSLA '87: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, 441–451, 1987.

[4] Carey, M. J., DeWitt, D. J., Of objects and databases: A decade of turmoil. *VLDB'96, Proceedings of the 22nd International Conference on Very Large Data Bases*, 3–14, 1996.

[5] Chamberlin, D., XQuery: An XML query language, *IBM Systems Journal*, 41(4), 597–615, 2002.

[6] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A.,

Gruber, R. E., Bigtable: a distributed storage system for structured data, *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 205–218, 2006.

[7] Cook, W. R., Ibrahim, A. H., Integrating programming languages and databases: What is the problem, *ODBMS.ORG, Expert Article*, 2005.

[8] CouchDB, `couchdb.apache.org/`, retrieved March 17, 2015.

[9] Dean, J., Ghemawat, S., MapReduce: Simplified data processing on large clusters, *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 137–150, 2004.

[10] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W., Dynamo: Amazon's highly available key-value store, *SOSP '07: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 205–220, 2007.

[11] Drepper, U., Parallel programming with transactional memory, *Communications of the ACM*, 52(2), 38–43, 2009.

[12] Grossman, D., The transactional memory/garbage collection analogy, *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, 695–706, 2007.

[13] Herlihy, M., Moss, J. E. B., Transactional memory: architectural support for lock-free data structures, In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 289–300, 1993.

[14] International Organization for Standardization, Database Language SQL, ISO/IEC 9075, 1986,1989, 1992, 1999, 2003, 2008.

[15] Java database connectivity, `www.oracle.com/technetwork/java/javase/jdbc/`, retrieved March 17, 2015.

[16] Java persistence api, `jcp.org/aboutJava/communityprocess/final/jsr317/`, retrieved March 17, 2015.

[17] Language integrated query, `msdn.microsoft.com/en-us/library/bb397926.aspx`, retrieved March 17, 2015.

[18] Maier, D., Representing database programs as objects, *Advances in Database Programming Languages*, 377–386, 1990.

[19] Meijer, E., Confessions of a used programming language salesman (getting the masses hooked on Haskell), *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 677–694, 2007.

[20] Russell, C., Bridging the object-relational divide, *Queue*, 6(3), 18–28, 2008.

[21] Scala. `www.scala-lang.org/`, retrieved March 17, 2015.

[22] Shavit, N., Touitou, D., Software transactional memory, *PODC '95: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, 204–213, 1995.

[23] Wadler, P., Databases and programming languages: Together again for the first time, *DBPL '11: Proceedings of the 13th International Symposium on Database Programming Languages*, 2011.