Exposing Undergraduate Students to Database System Internals

Anastassia Ailamaki
School of Computer Science
Carnegie Mellon University
natassa@cs.cmu.edu

Joseph M. Hellerstein
Computer Science Division
University of California, Berkeley
jmh@cs.berkeley.edu

Abstract

In Spring 2003, Joe Hellerstein at Berkeley and Natassa Ailamaki at CMU collaborated in designing and running parallel editions of an undergraduate database course that exposed students to developing code in the core of a full-function database system. As part of this exercise, our course teams developed new programming projects based on the PostgreSQL open-source DBMS. This report describes our experience with this effort.

1 Motivation

As background, we revisit the previous teaching experiences at Berkeley and CMU that led to our course design this year.

1.1 Undergraduate DBs at Berkeley and CMU

Berkeley has traditionally offered a strongly systems-oriented undergraduate database course, using the Ramakrishnan/Gehrke textbook. Throughout the semester, the course typically covers DBMS internals, traditional "user-level" issues like SQL and database design, and an overview on topics such as web-search (IR) techniques, XML, and object-relational features. Until last year, the course used the Minibase software from Wisconsin to expose two- or three-student groups to the software engineering challenges inherent in translating textbook algorithms into working code.

Carnegie Mellon started offering an undergraduate database course in Spring 1999. The material covered in the lectures was similar to that covered by the Berkeley course, although the Silberschatz/Korth/Sudarshan textbook was used. The course projects, however, focused mainly on data structures and SQL applications. Interested undergraduates were encouraged to enroll to the entry-level graduate database course (introduced in Fall 2001), that covers in-depth systems issues and selectively uses Minibase assignments as part of the course workload.

1.2 Prior Student Input

While at both schools the students' reaction to the courses was largely positive over the years, their feedback on Minibase was fairly negative. Although Minibase offers a significant code base and provides a challenging coding experience, the students complained that it was difficult to understand and extend, and yet at the same time also too "mini" – not representative of real-world software. At Berkeley, departmental student representatives singled out the database course for "constructive criticism" in their annual report at the faculty retreat *two years running*, advocating for removing Minibase from the curriculum. At the same time, CMU students were happy with the undergraduate course, but several wished for a more systems-oriented project that would offer them hands-on experience on the database internals.

2 The Berkeley/CMU Experiment

Anyone who has designed a student coding project knows that it entails unusual effort: designing the requirements and implementing the project, monitoring the student progress and repairing bugs on the fly, and designing and implementing test cases for grading the students' efforts. In Spring 2003, the Berkeley and CMU undergraduate database course staff collaborated in designing a curriculum that offers hands-on experience on the code of a real database system.

2.1 Designing the Course Schedule

To achieve synchronization, we agreed upon the same textbook (Ramakrishnan/Gehrke) and a similar schedule for the entire semester. The course structure was based upon a somewhat unusual organization that has been successfully used at Berkeley in the past:

A front-loaded semester. The "heavier", time-consuming projects are assigned in the first half of the course. Students are warned on day one that the busy implementation work in the beginning of the semester will be rewarded with "think time" later on. This structure

has proven to be very popular, since it balances both the students' and the TAs' load across the semester – most are busy with exams and projects from other courses toward the end of the semester, when the database course workload is light. In addition, the TAs' advisors appreciate their students' lack of conflicts at the end of the semester. The staff challenges with front-loading are to prepare the assignments on time (difficult coding assignments take longer to design) and to cover enough material by the time the assignments are passed out.

A bottom-up approach. In the first few lectures, the course touches only lightly on high-level issues like data models and query languages by briefly introducing the relational model and very simple SQL, contrasting it with the web documents and keyword search that students are familiar with. For the next several weeks, we dive into system implementation issues like access methods, buffer management, and join algorithms. After that, we begin to weave in deeper discussions of the relational model and languages, database design, normalization, and so on. As is not uncommon, we save transactional issues for later in the semester, when students have a good understanding of the entire "single-user" database problem space.

Apart from being necessary to front-load the semester, the bottom-up approach helps to stress the general applicability of storage, buffering, indexing, and query processing ideas for various information systems tasks, including specifically both SQL databases and web search engines. While few students will grow up and hack on the internals of a DBMS, more find themselves utilizing related ideas: buffers or caches, disk-based data structures, pipelined dataflow operators, dynamic programming, etc. Our colleagues were initially concerned that the bottom-up approach gives students very little data modeling context for the work they do early on, counting on their intuition and their willingness to "believe" that the context will come clear in later lectures. The students, however, have not complained about context weakness, as we take care throughout the semester to bindthe covered material together and emphasize the big picture.

2.2 Choosing PostgreSQL

We decided to use a full-fledged open-source DBMS for a number of reasons. First, the students at both schools were asking for a more "real" DBMS to work with, and this did not seem to pose a much bigger coding challenge than minibase. Second, the students would see a system supporting the full features of a standard relational DBMS, with a healthy exposure to the attendant complexities. Third, the students could play with the system *in action*, doing "real things"; Minibase only supported homework test drivers, and could not be used as a functional query system. Fourth, the students would wrestle with a system that they could feel to be "real": one that is in daily use in practical applications, discussed in active implementation forums, worth citing on their resumes, and architecturally and functionally similar to commercial systems.

We considered both of the leading open source systems: MySQL and PostgreSQL. MySQL has the advantage of significant opportunities for student extension, since it actually comes with almost none of the features taught in a typical DB systems course – it has no cost-based optimizer, no B+-trees, no fine-grained concurrency control, no recovery, no hash joins, etc.[1] By contrast, PostgreSQL already has most of the features usually taught in class. Of course, this it raises (surmountable) difficulties in inventing assignments where students can profitably extend the system with new features.

We finally chose PostgreSQL for a number of reasons. First, Berkeley still has extensive local expertise in the system, and CMU recently started to use it for research purposes. Second, because PostgreSQL is full-featured, students could consult the code to get a better sense of how pieces of a real system fit together (occasionally in lecture we showed PostgreSQL source code to highlight a point). Third, it felt inconsistent to teach about the importance of a full-featured database system and train the students using a limited environment such as MySQL.

2.3 Designing PostgreSQL Project Assignments

The joint Berkeley/CMU team designed two programming projects on "system internals", and one on "application design". These were supplemented with written homeworks covering SQL, normalization theory, and query optimization. Each programming project was done in a team of three students. The projects were as follows:

Buffer manager replacement policies. As a warmup assignment, students were given two weeks to (a) change PostgreSQL's buffer manager, replacing the default LRU page replacement policy with both CLOCK and MRU, (b) gather performance results of various queries that we provided over each of the three policies, and (c) explain their performance observations based on the conceptual discussion in class. The actual coding required for the assignment was minimal: a dozen or so lines of new code spanning one ".c" file and one header file, both of which we identified for them. However, the exercise exposed them to the task of debugging and validating modifications to a

[1] It should be noted that MySQL can be interfaced to better storage managers like BerkeleyDB and InnoDB for improved indexing, concurrency and recovery support. We felt that using more than one system was a poor option for educational purposes.

real system: understanding a client-server process architecture and the way that a debugger is used in that context, gathering performance traces, and mapping from experimental results to concepts from class.

Operators for query processing. In their most challenging project, the students had to add a new "iterator" to the query executor: a hash-based grouping operator, capable both of spilling to disk when necessary, and of maintaining memory-only performance for large inputs with few distinct groups. The algorithm was based on the Hybrid Cache scheme [1], modified to do grouping rather than function caching. The TAs made the necessary changes to teach the Postgres' optimizer to choose the new iterator. The students were given a naive main-memory-only hash group-by iterator as a starting point – already part of the latest PostgreSQL CVS repository – and were given three weeks to add the support for spilling to disk just when necessary.

The project involved understanding and implementing a fairly complicated query processing algorithm. Perhaps more challenging and useful was the required experience interfacing with pre-existing subsystems that aren't typically taught in the textbooks. In order to manage their hashtables, students had to understand a region-based memory allocator package – a ubiquitous subsystem in any real DBMS that is rarely discussed in classes. In order to manage the disk, they had to understand the temporary-relation I/O interfaces in PostgreSQL. In order to support aggregation in a generic way, they had to understand its structure as a triple of functions to initialize storage, accumulate tuples into an opaque temporary object, and finalize the result tuple. Finally, the data they were hashing, manipulating and storing was in PostgreSQL's native in-memory tuple format, and they had to understand the routines to work with that format. After getting the code working, they were again given a workload to run so they could measure the tradeoffs between hash-based and sort-based grouping, and explain these tradeoffs based on the conceptual discussion in class.

A web-based application. At Berkeley, students implemented a package-tracking application (*a la* UPS or FedEx) running off a live database server, whereas CMU students were asked to implement an application for University movies, in which users would register and post movie ratings and comments. For both applications, the students were given stock Apache and PostgreSQL servers, the database schema, and the HTML pages for the presentation of query forms and result pages. Their task was to write PHP scripts embedded into the HTML to connect to the database, translate the HTML forms into SQL queries, and iterate through result rows, printing them on screen. This 2-week assignment gave students the flavor of the kind of web-based application development that is a very common use case for database today.

2.4 Evaluation

At both schools, student feedback was quite positive. During the semester, it was easy to see both qualitatively and quantitatively how the PostgreSQL assignments were better received than Minibase had been in prior years. While there was still discussion of subtle coding points on the class newsgroup, these tended to focus on substantive system issues (e.g., the use of a region-based memory manager) rather than implementation details (e.g., the idiosyncratic exception handling mechanism in minibase). Joe Hellerstein received his highest average student rating for "Teaching Effectiveness" in five times teaching the course, and student representatives again singled out the course at the annual faculty retreat – this time to cheer the replacement of minibase with PostgreSQL. Natassa Ailamaki's ratings were also very high (no base for comparison, since this was her first undergraduate course). Naturally, it is difficult to separate these results from other factors – in particular, the TAs for the course this outing were especially dedicated. However, our feeling is that the use of the same "real" system for both "internals" assignments and for application assignments as well as the front-loaded course structure left the students with a unique sense of accomplishment.

2.5 Course Materials

Course materials – in particular the PostgreSQL project handouts – are available for viewing at `http://db.cs.berkeley.edu/dbcourse` and at `http://www-2.cs.cmu.edu/~natassa/15-415`. Further materials, including homework solutions and grading scripts, are available to instructors upon request.

Acknowledgments

The PostgreSQL internals projects were designed collaboratively by teams at Berkeley and CMU. The Berkeley team consisted of Mike Franklin, Joe Hellerstein, Ryan Huebsch, Sailesh Krishnamurthy, Boon Thau Loo and Li Zhuang. The CMU team consisted of Natassa Ailamaki, Spiros Papadimitriou, Minglong Shao, and Joe Trdinich. The CMU team cordially thanks Christos Faloutsos for his input and encouragement.

References

[1] J. M. Hellerstein and J. F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 423–424, Montreal, June 1996.