Teaching Statement

Peter Alvaro
Computer Science Division
University of California, Berkeley
palvaro@cs.berkeley.edu

Of all of my accomplishments in graduate school, teaching has been one of the most rewarding and challenging. The best memories of my own education are those in which topics were brought to life by exceptional instructors, and I hope one day to help students form such memories.

My background and interests equip me to teach undergraduate courses on databases and data management, distributed systems, operating systems, networking, software engineering, programming languages and logic. In addition to these core subjects, I am interesting in teaching graduate and advanced undergraduate courses on current research in distributed systems, particularly where it intersects with data management systems and languages.

Splitting one’s time between research and teaching can be challenging—both require our complete attention and our best efforts. Part of the problem for many graduate students is viewing the two pursuits as strictly independent. I believe that teaching is not merely an adjunct but a critical component of the research process. First, I do not feel as though I truly know a subject until I have lectured on it. Second, capturing and maintaining the attention and enthusiasm of an undergraduate classes challenges us to go beyond mastery of the material: we must provide context, motivation and energy to animate the presentation. Finally, once we have captured their interest, computer science students—who are skeptical by nature, and reluctant to accept an idea until they themselves have reasoned it out from first principles—make fantastic critics. If holes exist in your outline, students will find them; if errors exist in your theory, students (like flies to flypaper) will stick to them. If nothing else, an engaged but skeptical classroom (when we are at our best, we create such a classroom) prepares us for doubt and criticism, and teaches us to temper our enthusiasm for the subject with patience in our presentation of ideas.

Large-scale systems are difficult to design and build, and require organizational structures—like a parallelizable division of labor with centralized control and vision—that can be as difficult to maintain among students as they are among computers. Hence group work in systems classes is not just a pedagogical tool, but practical preparation for the collaboration, delegation and communication required to implement software systems in the large. Systems classes must teach computer science students how overcome the (laudable) impulse to understand every part of a complex system at the same level of detail. The tension between students’ desire for independence and the real-world constraint that software requires collaboration and hierarchy can be tricky to navigate, but is essential preparation for futures in software engineering.

Programming the Cloud

In the Fall of 2011 and Spring of 2013 I had the pleasure of co-teaching a course on distributed systems with my advisor Joe Hellerstein. The purpose of the course was to present fundamental distributed systems concepts (Berkeley did not offer any undergraduate courses in this area) through the lens of practical software development in the Bloom language.

I was responsible for co-designing the course content, creating and grading the homework and projects and presenting half of the lectures. Each week focused on a particular distributed systems theme, such as ordering, replication, concurrency control, consensus, or parallelism. On the first meeting of each week, Professor Hellerstein or I delivered a high-level lecture presenting the core problem and its historical context; in the second meeting, we led group programming exercises to embody the lesson in a reusable code module. For example, in Week 2 I gave a
lecture on order in distributed systems, presenting the asynchronous network model and the hierarchy of message delivery order constraints (FIFO, causal, total). In our second meeting that week, we broke into groups to play an adversarial game: devise a protocol that ensures FIFO delivery order of messages (slips of paper) between two endpoints, even though an adversary is allowed to arbitrarily delay (hide) or reorder (shuffle) messages. To my surprise (and delight), a variety of strategies emerged; as homework, students implemented their schemes in the Bloom language, to be reused later in the course. By the seventh week, students used the components they had developed each week to implement a quorum-replicated key-value store.

In addition to presenting core distributed systems concepts and participating in hands-on programming, we provided students with perspective on the state of the art beyond academia by regularly inviting industrial guest speakers. The speakers included database and distributed systems experts such as Doug Terry (Microsoft Research), Brian Cooper (Google) and Pat Helland (Salesforce), NoSQL architects and engineers from LinkedIn and Basho, and analysts from Twitter and Yammer. One of the stated goals of the course was to study, in the context of the Bloom language and its analysis tools, the “best practices” of experts in the field. These talks inspired students to explore the feasibility of enforcing design patterns like “ensure that updates to shared mutable state commute” at the level of language and analysis.

**Introduction to Database Systems**

In Spring of 2009 I served as a teaching assistant for cs186: Introduction to Database Systems, an upper-division undergraduate course that split over 100 students across just two teaching assistants. It covered all aspects of Data Management system design and implementation, including storage formats, buffer management, concurrency control, query processing and optimization and recovery, as well as query languages and the relational algebra and calculus. I designed and graded examinations, homework and projects and led discussions and exam preparation sessions.

The main course project involved adding a new feature to the PostgreSQL database: the index stride technique for online aggregation. Students were required first to orient themselves with a codebase of nearly a million lines. To implement the index striding feature, which visits indexed grouping sets in a round-robin fashion, the students had to make modifications and additions to the PostgreSQL parser, planner and executor. Several former students (now employed as software engineers) have indicated that this experience “hacking” a production-scale open source database (as opposed to implementing “toy” systems from scratch) was the best preparation of any university course for the realities of industrial software engineering.

The electrical and computer engineering honor society Eta Kappa Nu (HKN) has made the student reviews for this course public.

**Undergraduate Research**

I believe that encouraging undergraduates—even those who do not intend to earn a graduate degree—to take an interest in current computer science research is an often-overlooked duty for instructors. Because computer science is so focused on practice, CS undergraduates get few opportunities to engage with cutting-edge research. Reading conference papers can present great challenges but provides valuable rewards: we should strive to provide the necessary context to make conference papers accessible to undergraduates. Collaborating with graduate students on large-scale projects is more common, but mentorship is critical in such collaborations—otherwise, undergraduates provide cheap labor without being rewarded with real research experience.

A fruitful collaboration with an undergraduate should carry through to a research milestone such as a publication or grant proposal. One of my undergraduate research projects bore such fruit: in Spring 2012 my student Andy Hutchinson and I designed and built a declarative testing framework for the Bloom language, which we published and presented in the ACM DBTest workshop co-located with SIGMOD 2012. This was an ideal collaboration, in which the undergraduate participated in the end-to-end process: identifying and modeling a problem, designing and building a proof of concept, writing a research paper and ultimately participating in the presentation of the work to the community.