Our goal with Coursica was to provide the best mobile course browsing experience possible to a user of the app. To do that, we needed to incorporate data from the CS50 Courses API and from the Q data dump in a way that was easy and fast to interact with and gain valuable information from. Early on, we decided to forego most of the CS50 courses API's endpoints in favor of storing the data locally and performing our own searches and filters. This had two advantages. The first is that searching and exploring the course catalog on our app is virtually instant. The difference proves especially clear when internet connection is spotty, a common scenario for mobile users. The second advantage was that it gave us total control over the filters and searches of the data we could allow. We were able to use this freedom to add a new layer to exploring Q data. We're proud that our users can search ranges of all of the categories of Q data, which isn't possible even on the CS50 courses website or through the API.

Technically, we decided to implement our app on top of Core Data in iOS. Core Data is a MySQL database under the hood, which uses what are called NSPredicate objects to execute fetch requests on a persistent store of data on the hard drive. This matched our needs of fast, local searching. We defined a database schema in Coursica.xcdatamodeld, and then used AFNetworking to pull down all of the course data from the CS50 courses API when the user opens the app. We used an open-source library called CHCSVParser to ingest the Q data dump into our Core Data store, and calculate Q scores for every class. After this initial setup, the app is extremely fast, even when performing complex searches and filters of the thousands of courses. Core Data provides a helpful framework for interacting with a large database stored on the hard drive that we used extensively. It achieves this speed and memory efficiency because it only loads into memory exactly the pieces of data that are needed. Still, it can be slow when it does have to retrieve data from the disk. Much performance tuning we did revolved around minimizing the number of times we had to retrieve data from the disk. One example is in the Courses.m file, in the updateCourses: method where most of the initial setup of the database happens. Our huge initial data import took much longer initially, before we realized we were making individual fetches on the database which are extremely slow. Retrieving the q score data and comments for every single course in the database took a very long time. We sped this up by loading the numerical Q score data into memory without the comments, and loading it all in one fetch, storing it in an NSDictionary (a hash table, internally). This let us access the Q scores in close to constant time without the overhead of a fetch on the hard drive.

Correlating comments with the courses they belonged to is a very expensive operation because of the size of the comments database. So, we decided to only do this on-demand, when a user specifically presses the button to view comments within one class. Similarly, we store the averaged Q scores in the course object in the database so we can search quickly without having to recompute this information.

One of our greatest frustrations with the CS50 courses website, which we thought we could improve, is the instability and slowness of search. Results sometimes vary and defy explanation, and usually are processed slowly. Our search and filters work on a string of NSPredicate objects that represent the filter to be applied by the app on every course in the database. When a search is executed, Coursica walks down the predicate string and applies each filter in order. If an object fails one of the filters, non of the others are tested. So, we structured our search queries so that the fastest filters come first. For instance, search is by far the most computationally intensive. It has to do several string comparisons on every course object, and on several corresponding faculty objects for each course. On the other hand, the boolean comparison of whether a course is a graduate or undergraduate course is practically instant for a computer. So, we ordered our search and filter predicates by their complexity. This way, Coursica won't waste time searching through the faculty strings of a course if it can more easily eliminate the course because it is for graduate students and the user is an undergraduate.

Overall, we focused our development on speed and ease of use, especially when internet access may be slow or nonexistent. To realize these goals we chose to use Core Data, optimized our fetching of data from the disk, and optimized our searching methods.