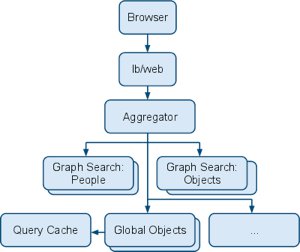
In fall of 2009, some of us at Facebook imagined a more interactive search experience, where high quality results would appear as the user typed. We summarized the vision in [a blog post](http://blog.facebook.com/blog.php?post=369584442130) introducing our new search box.  
  
Making this system real for 400 million users has been challenging at every level of the software stack. On the front-end, code running in users' browsers must consume and render results quickly enough to not distract the user as they type. We set a strict goal for ourselves that the new typeahead can't be slower than the existing typeahead for finding your friends, so every millisecond delay matters. Despite these performance constraints, the UI cannot be too minimalist; each result needs enough contextual clues to convey its meaning and relevance. Because the new typeahead auto-selects the first result when you hit Enter in the search box, we need near-perfect relevance so that the first result is always the one you're looking for.  
  
To satisfy all these constraints, we designed an architecture composed of several different backend services that were optimized for specific types of results. Let's follow a session as the request leaves a user's browser and becomes a set of navigable results.   
Architecture of Typeahead Search  
**1. Bootstrapping Connections**

As soon as the user focuses in the text box, we send off a request to retrieve all of the user's friends, pages, groups, applications, and upcoming events. We load these results into the browser's cache, so that the user can find these results immediately without sending another request. The old typeahead did this, but stopped here.

**2. AJAX Request**

If there are not enough results in the current browser cache, the browser sends an AJAX request containing the current query string and a list of results that are already being displayed due to browser cache. Our load balancer routes the request to an appropriate web machine. 

**3. Aggregator Service**

The php AJAX endpoint is a thin wrapper around a [Thrift](http://blog.facebook.com/blog.php?post=2261927130) service for handling typeahead queries. This service, called the "aggregator", is at the root of a tree of separate search services. Aggregators themselves are stateless and have no index of their own. They are instead responsible for delegating queries to multiple lower-level services in parallel and integrating their results. 

**4. Leaf Services**

In parallel, several different backend search services scour their indices for results that prefix-match the contents of the search box. Each leaf service is designed to retrieve and rank results on only a few specific features. The architecture of the system allows us to add new services, too, as we discover important sources of results and features.   
  
The *global* service maintains an index of all the pages and applications on the site. Unlike most of what we do at Facebook, these results are global because they contain no personalization. The signals that this service records are identical for all user on the site; for example, we might rank applications by the number of users who have interacted with them in the last month, and pages by the structure of the graph surrounding them. Since this service's results are independent of the querying user, we can save latency by storing recent results in a [memcached](http://www.facebook.com/note.php?note_id=39391378919)-based query cache.  
  
The *graph* service returns results by searching the user's neighborhood of the [graph](http://developers.facebook.com/docs/api). For some types of queries, a user and her friend's graph connections are a powerful signal of expressed preferences, and thus relevance. Graphs are notoriously difficult mathematical objects for computers to come to terms with: they are large, and are a minefield of computationally hard problems that appear simple. Our graph contains 400 million active users, and many billions of connections, both among users and from users to objects of other types: pages, applications, events, open graph nodes, etc.

**5. Merging Results**

The aggregator merges the results and features returned from each leaf service and ranks the results according to our model. The top results are returned to the web tier.

**6. Fetching Data and Validating Results**

The results returned by the aggregator are simply a list of ids. The web tier needs to fetch all the data from memcache/[MySQL](http://www.facebook.com/MySQLatFacebook) to render the results and display information like the name, profile picture, link, shared networks, mutual friends, etc. The web tier also needs to do privacy checking here to make sure that the searcher is allowed to see each result.

**7. Displaying the Results**

The results with all the relevant data are sent back to the browser to be displayed in the typeahead. These results are also added to the browser cache along with the bootstrapped connections so that similar subsequent queries don't need to hit the backend again.

**Putting it all Together**

Once the basic architecture was in place, we spent a lot of time getting everything to a production-ready state. We dark-tested by changing the old typeahead to send a query on every keystroke. We knew that usage patterns could dramatically shift once the new typeahead was fully launched, but this was a good first step to uncovering any scalability issues we might have. As a result of this testing, we discovered network limitations that caused us to adjust the topology of the architecture.  
  
We also experimented with a large number of user interface variants using A/B testing and more qualitative usability studies to help us make some thorny decisions. 

* Number of results: We wanted to show enough entries so that you would serendipitously stumble upon fun results while typing, yet showing fewer results would be faster and be less distracting. Our ultimate design varies the number of results shown based on your browser window height.
* Searching: Even though we made the Enter key auto-select the first result, we wanted to ensure users who wanted to search still could. We experimented heavily with the placement, wording, and appearance of the “See More Results” link that takes you to our traditional search page.
* Mouse vs. keyboard: While Facebook employees are heavy keyboard users, we discovered most users prefer to use the mouse to select from the typeahead. This led us to focus on mouse use-cases more than we naturally would have.

Other experiments included different picture sizes, ways of distinguishing between different result types like applications vs. people, highlighting the query string in results, and various ways of displaying mutual friends.  
  
As with all interesting engineering projects, the trade-offs are where we had the most challenges, surprises, and fun. Many technical decisions in search boil down to a [trilemma](http://l.facebook.com/l.php?u=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FTrilemma&h=VAQGNQgwo&s=1" \t "_blank) among performance, recall, and relevance. Our typeahead is perhaps unusual in our high priority on performance; spending more than 100 msec to retrieve a result will cause the typeahead to "stutter," leading to a bad user experience that is almost impossible to compensate for with quality. "Real-time search" is often used to mean chronologically ordered, up-to-date [content search](http://blog.facebook.com/blog.php?post=115469877130), but our typeahead search engine is "real-time" in the [older sense](http://l.facebook.com/l.php?u=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FReal-time_computing&h=LAQHFqaP2&s=1) of the term: late answers are wrong answers.  
  
We have a lot of refinement ahead of us. Our result page is still there, and probably always will be: there will always be a "long tail" of queries that are too heavyweight for our typeahead. But we are excited to have shipped a first version, and look forward to seeing how far we can push our typeahead into the territory currently occupied by result pages.  
  
*Keith Adams is a Facebook engineer in search, in search of new Facebook engineers.*