

# Indexing and ranking in Graph Search

## Search Ranking

Search ranking is the process of choosing the best set of results for a search query. Our team combines data-driven techniques with intuition to create a process that is roughly:

1. Come up with an idea for a ranking change – ideas come from a mix of creativity on the part of our ranking engineers and feedback from our users
2. Implement the ranking change, test it, and launch it to a very small fraction of our user base
3. Measure the impact of the ranking change

Our goal is to maximize searcher happiness, which we do our best to quantify through metrics like click through rate (CTR), NDCG, engagement, etc. We have to measure the impact of ranking changes on all of these metrics to maintain a proper balance.

We need tools and frameworks to facilitate this (i.e., we need a search engine). As we talked about in earlier posts, we have decided to use Unicorn as the fundamental building block of our search engine. Here we describe how we extended Unicorn with search ranking capabilities.

## Extending Unicorn to be a Search Engine

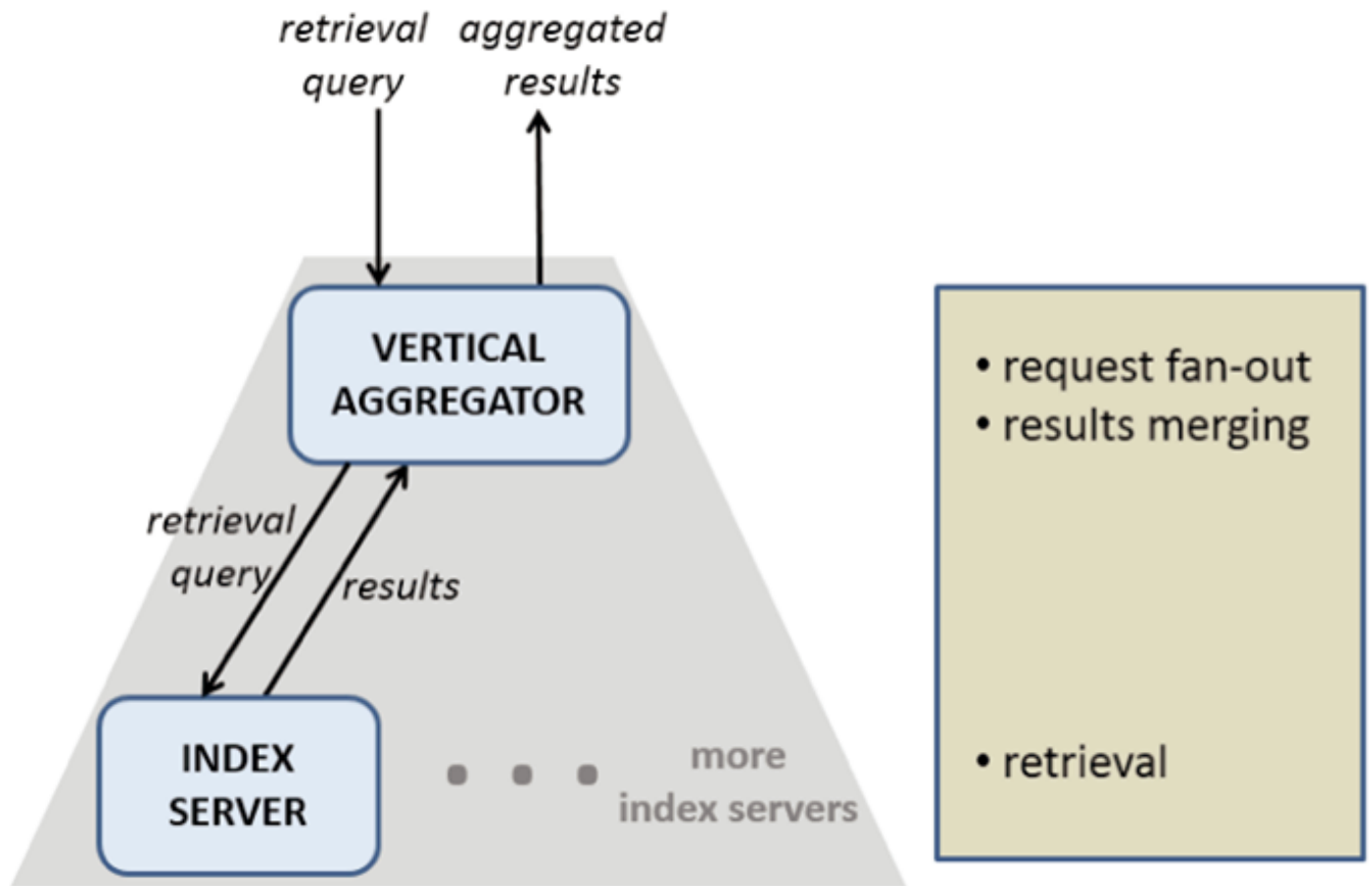
Unicorn is an inverted index framework and includes capabilities to build indices and retrieve data from the index. Some examples of retrieval queries described in our earlier post are:

(term mark)

(and david johnson)

(or zuck randi)

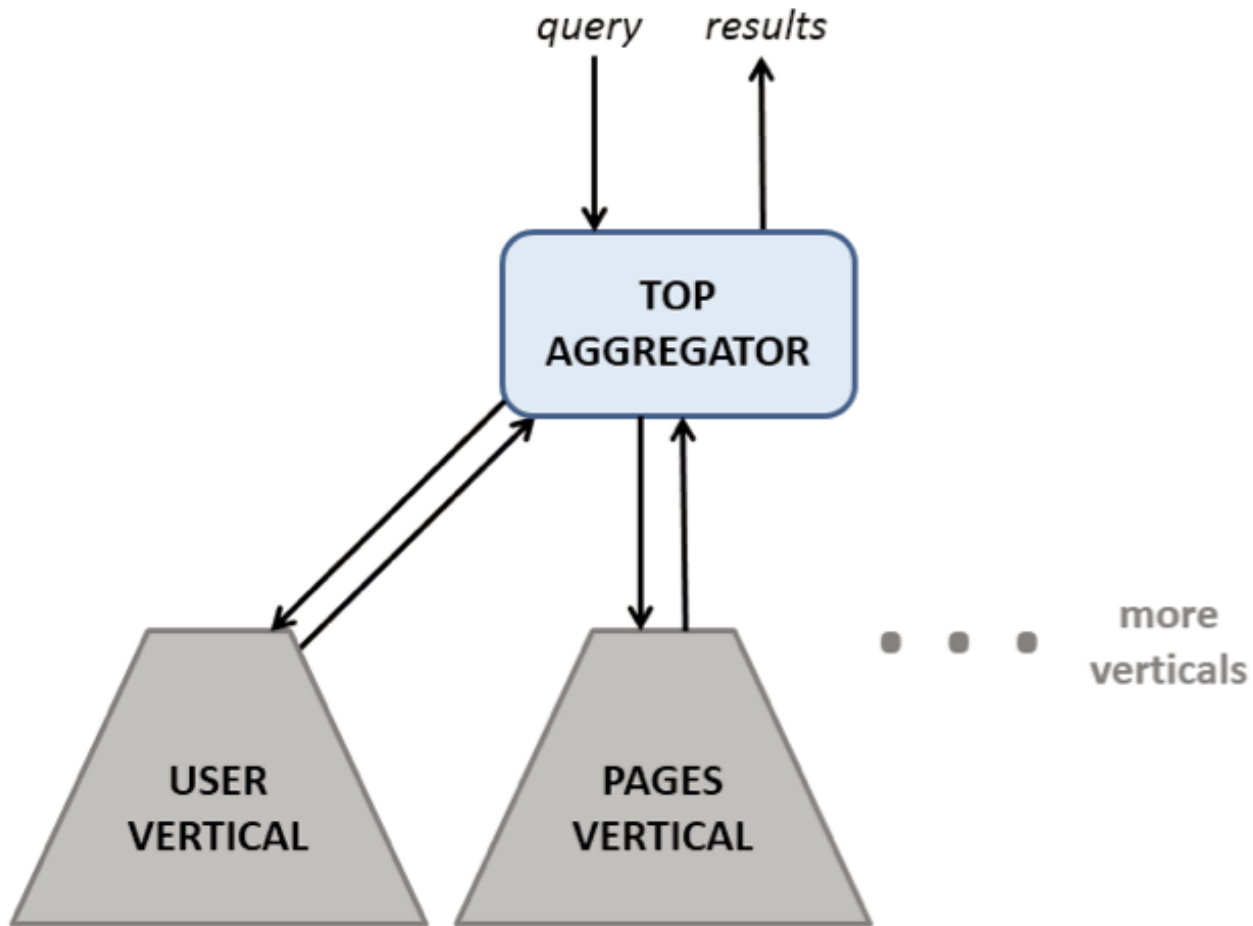
A Unicorn instance (or vertical) may store an index that is too large to fit into the memory of a single machine. So we break up the index into multiple shards such that each shard can fit on a single machine. These machines are called index servers. Queries to Unicorn are sent to a Vertical Aggregator, which in turn broadcasts the queries to all index servers. Each index server retrieves entities from its shard and returns it to the Vertical Aggregator – which then combines all these entities and returns them to the caller.



Unicorn is fundamentally an in-memory “database” with a query language for retrieval. Before we could use Unicorn for Graph Search, we had to extend it with search ranking capabilities. Here are some of the things we did:

***Keep different entity types in separate Unicorn verticals***

Given that different entity types (users, pages, photos, etc.) have very different requirements for ranking, we decided to maintain each entity type in a separate Unicorn vertical and create a new entity--the top aggregator--that would coordinate activities across the verticals.



### ***Extending the retrieval operations with weak “and” and strong “or”***

There are many instances (especially for social and contextual retrieval) where it is useful to weaken the semantics of “and” to allow some misses, and to strengthen the semantics of “or” to require certain matches.

### ***Query rewriting***

The queries that are issued by our end users are (obviously) not in the format required for retrieval. So we need to rewrite the query to translate the user intent and their social context into a structured Unicorn retrieval query. This is where we try to understand the query intent, bias it socially, correct spelling, use language models for synonyms, segmentation, etc.

### ***Scoring***

By default, Unicorn returns results in their static rank order--which means the relevance of the result for the query is not considered. Scoring is the step that takes place immediately after retrieval when a numeric score is assigned to an entity using both the entity data and the query.

### ***Forward index***

Additional information about the entity (beyond what is in the inverted index) is typically required for scoring. We added the ability to store a blob of metadata for each entity in the index. The indexing pipeline facilitates the creation of forward index data along with the inverted index.

### ***Result set scoring***

Scoring works on a single entity at a time and the score assigned is independent of the scores assigned to other entities. This can cause a result set to become very one-dimensional and offer a poor search experience, (for example, “photos of Facebook employees” may return too many photos of Mark Zuckerberg). Result set scoring offers yet another layer of filtering that looks at a number of entities together and returns a subset of these entities that are most interesting as a set (and not necessarily the highest scoring set of results).

### ***Blending***

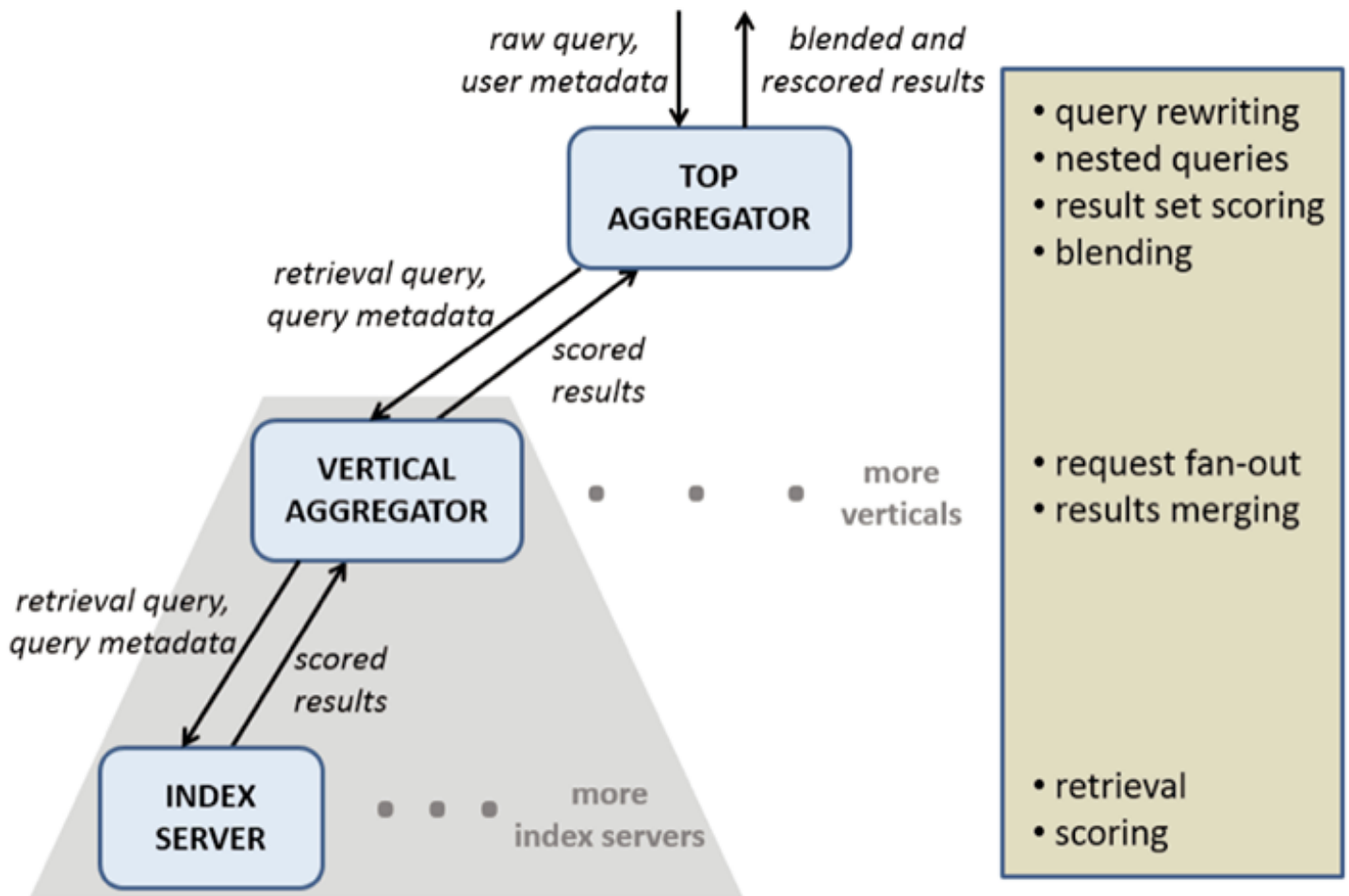
When results come in from different verticals to the top aggregator, they need to be combined into a single result set. To do this, the top aggregator needs to blend the results together in the right order. This requires normalizing the scores from different verticals so they can be compared against each other.

### ***Nested queries***

For many Graph Search queries, we have to perform multiple searches across many verticals and “join” the results. For example, “restaurants liked by Facebook employees” requires a search on the user vertical to obtain Facebook employees, and then a search with these users on the places vertical to obtain restaurants liked by them. Although Unicorn provides native support for nested queries through the “apply” operator, we had to extend that with a framework to inject different scorers (and result set scorers) for the different searches, and to allow for these scoring components to share data with each other.

### ***A/B Testing***

We built a framework to run ranking experiments where the happiness metrics are compared between various experiments against their controls. The Unicorn stack, with the components described above, is shown in the diagram below. The Top Aggregator maintains a separate query rewriter and a separate result set scorer for each vertical.



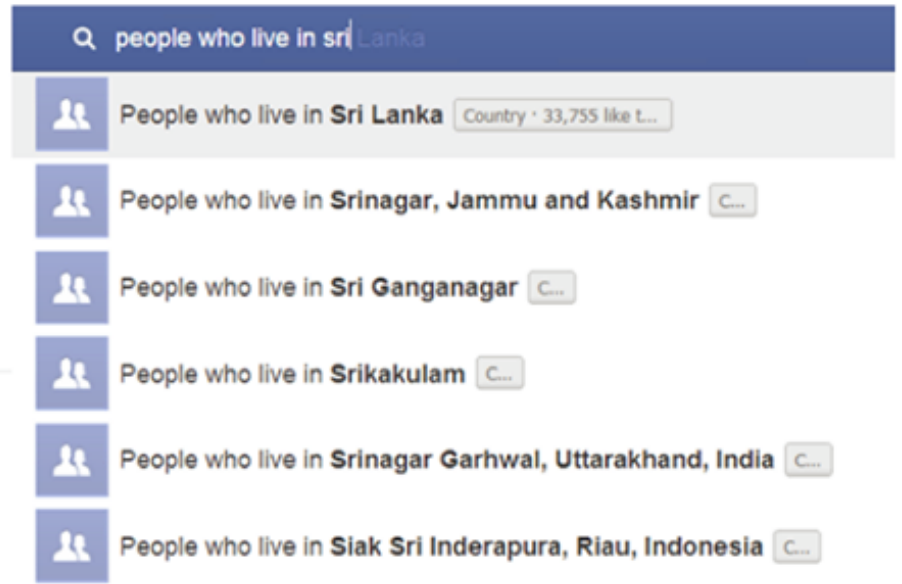
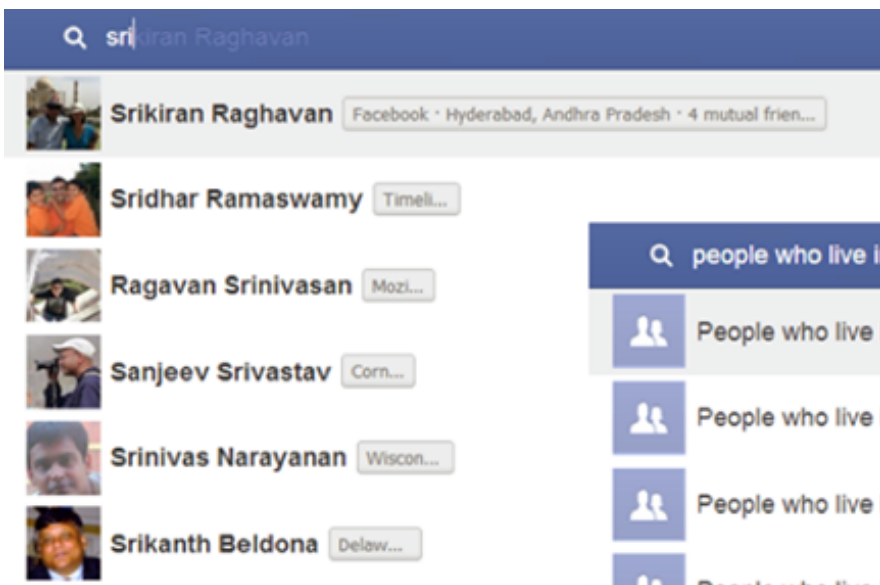
## The Life of a Graph Search Query

The interaction with Graph Search takes place in two distinct phases – the query suggestion phase, and the search phase. In the query suggestion phase the searcher enters text into the search box and obtains suggestions. The search phase starts from the selected suggestion and produces a results page.

### The Query Suggestion Phase

When a searcher types a query into the search box, a Natural Language Processing (NLP) module attempts to parse it based on a grammar. It identifies parts of the query as potential entities and passes these parts down to Unicorn to search for them. The NLP module also provides hints as to what kind of entity that may be – for example, if the searcher types “people who live in Sri”, the NLP module sends the query “sri” to Unicorn suggesting a strong bias towards cities and places. This causes results like “Sri Lanka” and “Srinagar” to be returned instead of results like “Sriram Sankar.”

Generally the NLP module also sends the entire query to Unicorn and when these results rank high, Graph Search works like Typeahead. The following screenshots show these two cases of calling Unicorn with the query “sri.”



In the query suggestions phase, the Unicorn Top Aggregator fans out the search request to all the verticals and then blends together the results on the way back. The sequence of steps that takes place are:

1. The NLP module parses the query, and identifies portions to search in Unicorn. Steps 2 through 9 are then performed simultaneously for each of these search requests.
2. The Top Aggregator receives the request and fans it out for each vertical. Steps 3 through 8 are performed simultaneously for each vertical.
3. The Top Aggregator rewrites the query for the vertical. Each vertical has different query rewriting requirements. Rewritten queries are typically augmented with additional searcher context.
4. The Top Aggregator sends the rewritten query to the vertical – first to the Vertical Aggregator, which passes it on to each of the Index Servers.
5. Each Index Server retrieves a specified number of entities from the index.
6. Each of these retrieved entities is scored and the top results are returned from the Index Server to the Vertical Aggregator.
7. The Vertical Aggregator combines the results from all Index Servers and sends them back to the Top Aggregator.
8. The Top Aggregator performs result set scoring on the returned results separately for each vertical.
9. The Top Aggregator runs the blending algorithm to combine the results from each vertical and returns this combined result set to the NLP module.
10. Once the results for all the search requests are back at the NLP module, it constructs all possible parse trees with this information, assigns a score to each parse tree and shows the top parse trees as suggestions to the searcher.

## The Search Phase

The search phase begins when the searcher has made a selection from the suggestions. The parse tree, along with the fbids of the matched entities, is sent back to the Top Aggregator. A user readable version of this query is displayed as part of the URL. For example, the query “restaurants liked by Facebook employees” displays the following URL:

273819889375819/places/20531316728/employees/places-liked/intersect

Here 273819889375819 is the fbid of the category “restaurants,” and 20531316728 is the fbid of Facebook. So the query says “intersect places of category restaurants with places liked by employees of Facebook.”

Once the Top Aggregator receives this query, it creates a query plan calling verticals sequentially--the first vertical is called, then the second vertical is called with a query constructed using results returned by the first vertical--and so on. In this particular case it sends the query to the user vertical with the following query:

(term employee:20531316728)

Suppose this returns results fbid1, fbid2, ..., fbidn. Then the places vertical is called with a query that looks like:

(and place-kind:273819889375819 (or liked-by:fbid1 liked-by:fbid2 ... liked-by:fbidn))

The results returned from the places vertical are the final results for the query. In more detail, the sequence of steps that take place for this particular query are:

1. The selected parse tree is sent to the Top Aggregator, which forms a query plan.
2. The query “(term employee:20531316728)” is prepared for the user vertical.
3. The Top Aggregator rewrites this query for the user vertical. The rewritten query is augmented with additional searcher context.
4. The rewritten query is sent to the Vertical Aggregator of the user vertical which passes it on to each of its Index Servers.
5. Each Index Server retrieves a specified number of entities from the index.
6. Each of these retrieved entities is scored and the top results are returned from the Index Server to the Vertical Aggregator.
7. The Vertical Aggregator combines the results from all Index Servers and sends them back to the Top Aggregator.
8. The Top Aggregator performs result set scoring on the returned results.
9. The returned results (fbid1, fbid2, ..., fbidn) are prepared for the places vertical as the query “(and place-kind:273819889375819 (or liked-by:fbid1 liked-by:fbid2 ... liked-by:fbidn)).
10. The rewritten query is sent to the Vertical Aggregator of the places vertical which passes it on to each of its Index Servers.
11. Each Index Server retrieves a specified number of entities from the index.
12. Each of these retrieved entities is scored and the top results are returned from the Index Server to the Vertical Aggregator.
13. The Vertical Aggregator combines the results from all Index Servers and sends them back to the Top Aggregator.
14. The Top Aggregator performs result set scoring on the returned results.

15. The resulting set of results is shown to the searcher.

 Restaurants liked by Facebook employees



Facebook Culinary Team

Restaurant · Cafeteria · ★★★★★

Mark Zuckerberg and other Facebook employees like this

Founded 5/08/2008. To date we have served 4,976,684 meals.

69 like this

Tom Stocky, Lars Eilstrup Rasmussen and 160 other friends like this

Like

Map

Q



Philz Coffee At Facebook

Coffee Shop · \$ (0-10) · ★★★★★

Henry Bridge and other Facebook employees like this

Philz Coffee was founded in 2003 by a man named Phil Jaber. Philz Coffee ha...


7:30 am - 5:30 pm

Gintaras Woss, Chris Bray and 39 other friends like this

Like

Map

Q



The Counter - Palo Alto

Diner · \$\$ (10-30) · ★★★★★

Kang-Xing Jin and other Facebook employees like this

Build your own gourmet burger from the bun up! You choose beef, chicken,...


11:00 am - 10:00 pm · (650) 321-3900

Tom Stocky, Chris Marra and 10 other friends like this

Like

Map

Q



Nopasf


Restaurant · ★★★★★

Samuel W. Lessin and other Facebook employees like this

nopa is a San Francisco gathering place north of the Panhandle, serving urba...

5:00 pm - 1:00 am · (415) 864-8643

More Than 100 Places



North America 24 / 24

REFINE THIS SEARCH

Place Type

Restaurant

Category...

Liked by

Facebook employees

Add

Name

Add...


Places in

Add...

Visited by

Add...

EXTEND THIS SEARCH



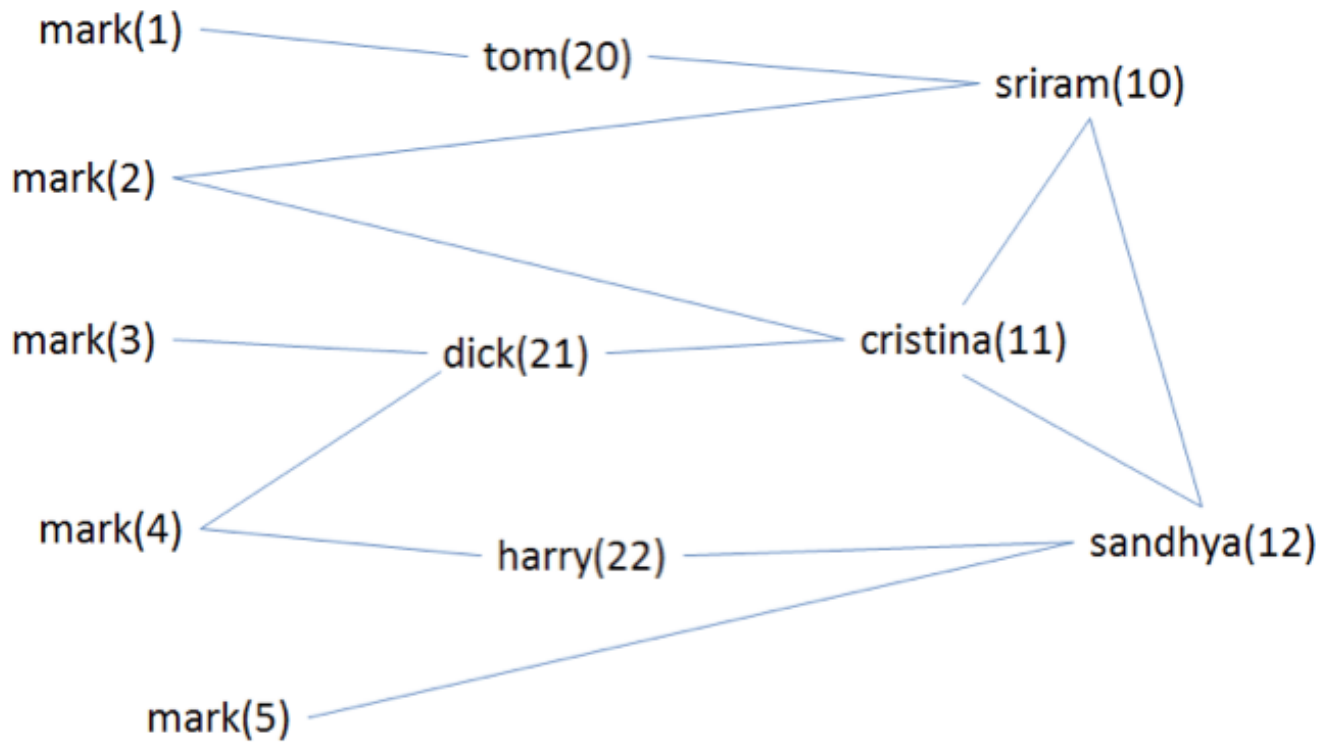
Photos from these places

## Social Query Rewriting

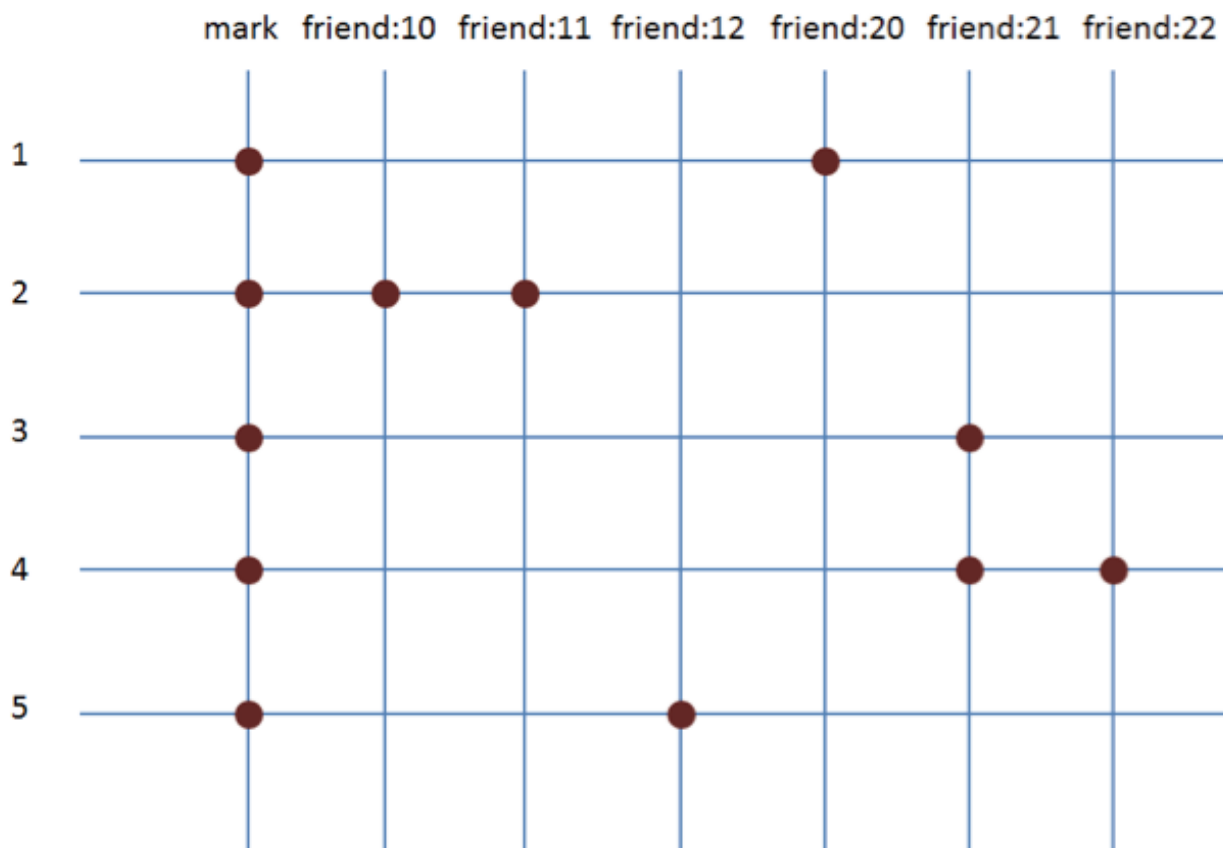
The Unicorn queries shown so far only reflect the retrieval constraints imposed by the search query. In addition, the query has to be rewritten to bias it socially and contextually for the searcher. The weak “and” and the strong “or” retrieval operators are used to achieve this. Let me continue with the example from the earlier post to illustrate how different searchers looking for “mark” end up getting the most relevant Marks.

Consider the following (hypothetical) snippet of the Facebook Graph where the edges are friendships, and the numbers in parentheses are fbids:





We index every user with their friends. In the diagram below, we show the index extended with the friend information for the 5 Marks in the above graph:



We rewrite the queries (to search for “mark”) to include the searcher and their friends as shown below:

Cristina searching for Mark:

(and mark (or friend:11 friend:10 friend:12 friend:21))

Sandhya searching for Mark:

(and mark (or friend:12 friend:10 friend:11 friend:22))

The rewritten queries shown above return all “marks” that are either friends or friends of friends of the searcher.

While these queries do well in returning friends and friends of friends called “mark,” they do not return any other results – for example famous people called “mark”. This is solved using the weak “and” operator that permits its operands to be missing some of the time:

(weak-and mark (or friend:11 friend:10 friend:12 friend:21)[0.7])

Here the second argument of the weak “and” only needs to be present 70% of the time. Now let us look at how we rewrite the Unicorn query that goes to the places vertical for “restaurants liked by Facebook employees”:

(and place-kind:273819889375819 (or liked-by:fbid1 liked-by:fbid2 ... liked-by:fbidn))

To make this query social and contextual for me, we would rewrite it further as follows (assume my friends have fbids 11, 12, and 20, and assume that the Facebook Graph indicates that I am in the Palo Alto area):

(weak-and

(and place-kind:273819889375819

(or liked-by:fbid1 liked-by:fbid2 ... liked-by:fbidn))

(strong-or

(or liked-by:11 liked-by:12 liked-by:20) [0.4]

(or location:palo-alto) [0.7] ) [0.6]

)

The above query requires 60% of the results to be either liked by me, or my friends, or be located in Palo Alto.

### **Not Only Finding the Data, But Continuing to Make it Useful**

Generally when people talk about search, they immediately mention machine learning and scoring. Scoring is an important part of the search ranking pipeline, and it is the part that can be automated using machine learning.

Machine learning (as it applies to search) is the process of automatically coming up with a scoring function using data from lots of previous search queries and the actions taken on them. We build features that become inputs to the scoring function – features are developed using our domain knowledge and quantify important properties of the entities and/or the query. Examples of features are: the distance between a place and the searcher; the closeness of a user results from the searcher in terms of friend connections; the amount of overlap of the query string with the entity name; etc. Although machine learning frameworks can use many thousands of features, we have decided to limit the number of features and engineer these features well. We use machine learning frameworks that combine features in a simple manner – as a linear weighted combination of the feature values. This allows us to keep the scoring formula understandable and we are able to make manual tweaks to the trained formulae as necessary.

The retrieval operations allow for diversity during retrieval using the weak “and” and strong “or” operators. In

order to maintain this diversity, we support diversity during scoring by Unicorn offering the ability to plug in multiple scorers for the same search query. This means an entity may have multiple scores assigned to it. We then have the ability to select the top results with respect to each of these scores. For example, we may decide to have three scoring functions for ‘nearby’ search – one that is based on overall popularity, one that is social, and one that gives lots of weight to distance. We can then come up with a policy to have a certain number of results with respect to each function and let the result set ranking do the final arbitration on which results to send up to the caller. We believe that a simple scoring framework along with additional support for diversity results in a more straightforward scoring framework that can be understood, maintained, and debugged most easily.

## Going Forward

Now that we have launched Graph Search, we are learning from the usage to understand which queries are popular and how they need to be optimized. We are also extending our search capabilities to do better text processing and ranking and have better mobile and internationalization support. Finally, we are also working on building a completely new vertical to handle searching posts and comments.

While we have achieved a lot with the launch of Graph Search – by growing our index to hundreds of billions of nodes and trillions of edges – we have in many ways only scratched the surface in building a comprehensive search engine over the Facebook Graph. As Graph Search grows and we add more features, more people will optimize their data for the Graph Search engine (thus improving the quality of the Facebook Graph), leading to an even more awesome search experience. The next few months and years are going to be very exciting!

Excerpted from *Under the Hood: Indexing and ranking in Graph Search*  
<https://www.facebook.com/notes/facebook-engineering/under-the-hood-indexing-and-ranking-in-graph-search/10151361720763920>

READABILITY — An Arc90 Laboratory Experiment <http://lab.arc90.com/experiments/readability>