Unicorn: A System for Searching the Social Graph

The domain area of Unicorn is social graph search and retrieval. Unicorn is a in-memory indexing system developed by Facebook Inc. used for social graph retrieval and social ranking. It is built to support trillions of edges, billions of users and thousands of servers. Unicorn also supports real-time updates, per-edge privacy (even though privacy is enforced by the front-end) while serving billions of queries each day at low latencies. It was developed with the goal of being able to quickly and scalably search the social graph and perform complex set operations.

According to [Curtiss et al.] there exist similar search indexing systems, but no one that is initially built to support social graph search and with the scale of Unicorn both in data volume and query volume. Hence, the motivation behind Unicorn was to build a indexing system specific for social graph purposes. By building a specialized indexing system for the Facebook back-end, Facebook-specific properties could be incorporated into the design. Hence, choices such as representing the social graph as an adjacency list because of observations that the social graph is sparse (average number of friends is 130) The paper says that the motivation is in section 2-5, but I'm not sure if this should rather be in the "techniques section".

Most of the paper goes into describing query technique, so there is no room to go into details here, but the main techniques developed will be presented. In the data model, each document has a sort key, to ensure that the globally most important id's are stored at the top of the adjacency list. However, global relevance is not enough to provide good query results, therefore extra data is stored with each document to provide useful filtering options. Another technique presented is that Unicorn is sharded by result-id. This ensures that even though some machines might be down, the system is still able to return some results. Sharding also provides the advantage that set operations can be performed at index server (leaf) level, so that the computational load can be split between a number of machines. To ensure relevant query result uses two operators called WEAK-AND and STRONG-OR. The first one forces some fraction of the result to possess some trait, while as the second one requires certain operands to be present in a fraction of the results. Together, these operators help provide the user with more relevant results. Unicorn provides scoring based on meta data, but also implement a heap-based data structure to ensure diversity in the results by prioritizing results that provide diversity to the results. The Unicorn cluster is partitioned into verticals each representing an entity (users, pages, etc..). The top aggregator decides which vertical(s) the query is sent to. A vertical consists of Unicorn tiers containing racks which each hold one rack aggretator and index servers. An advantage of this partitioning is that optimal ranking algorithms can be built for each vertical. Results are collected, combined from the verticals and truncated if necessary. Two operators that Facebook is very proud of is their APPLY and EXTRACT operators. APPLY allows clients to

query for a set of ids and then use those to construct and perform a new query, the advantage of this operator is that it allows the system to perform expensive operations at a low level. The extract operator performs partly denormalization of data to save space. As mentioned above, the front-end takes care of privacy, but in order to be able to do so, Unicorn provides the front-end with the lineage of the data. By lineage, they mean the relevant information about the edges traversed for finding the results. By inspecting the lineage, the front-end can filter based on privacy settings.

Since the main focus of the paper is to describe the data model and query language as well at the evolution of Unicorn it do not include many results. Even so, the few result mentioned will be presented here. To get an understanding of query performance, the paper includes performance results for the query "People who like Computer Science". Using their APPLY operator the query would look like this

(apply friend: likers: 104076956295773)

where **104076956295773** is the id of computer science, **likers:** means get the likers of, and **friend:** means get the friends of.. Since the inner query will return 6 million likers, it has to be truncated before executing the out query. By varying the inner truncation limit the latency and average aggregate CPU time could be evaluated. Their results shows how large truncation limits leads to higher latency and cost. As the inner truncation limit reached 5000, the latency was passing $100ms$ which they consider to be a reasonable threshold. These results show that truncation will always be required, and that it is therefore very important with good inner ranking, even though needle-in-a-haystack results are better than nothing.

The application of Unicorn is, of course, to be the back-end of Facebook's search engine. Unicorn should however be possible to use for different types of Internet graph search where nodes represent entities and edges represents relationships between those entities. As mentioned earlier, If I remove it from the paragraph above I need to rewrite this. Unicorn is implemented to optimize search in the Facebook social graph, and therefore lots of design choices are based on Facebook-specific properties. Unicorn would therefore be most useful for applications where the graph structure, data model, and query logic is similar.

There is no doubt that Facebook is an extremely successful company, and it is very hard to criticize a system which so many people love to use everyday, myself included. However, I do have some criticism when it comes to how Unicorn is presented in the paper. The paper is written more as a sales-pitch for Unicorn than an actual research paper. Although it briefly mentions other systems in the "Related Work" section it does not compare its performance with any of these systems. When explaining different angels of trying to implement friends-of-friends efficiently for example, they only compare their proposals to

the most basic, brute-force solution, of storing friends-of-friend as a separate list for each user. I also miss an acknowledgement of short-comings and future improvements. As an example, they briefly mention that letting the front-end deal with privacy filtering imposes an modest efficiency penalty, but then they jump right to defending this choice, without any deliberation about how this could be handled differently. Despite these remarks, Unicorn is a system developed for a huge real-world application, to support changing needs and growing scale and it works. Researches working for such an company do not have all the time in the world to implement and compare hundred different systems and compare their performance, their job is to supply costumers with what they need and want when they need it, and that is what they do.