Unicorn: A System for Searching the Social Graph

Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko,
Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin,
Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, Ning Zhang

Facebook, Inc.

ABSTRACT

Unicorn is an online, in-memory social graph-aware indexing system designed to search trillions of edges between tens of billions of users and entities on thousands of commodity servers. Unicorn is based on standard concepts in information retrieval, but it includes features to promote results with good social proximity. It also supports queries that require multiple round-trips to leaves in order to retrieve objects that are more than one edge away from source nodes. Unicorn is designed to answer billions of queries per day at latencies in the hundreds of milliseconds, and it serves as an infrastructural building block for Facebook's Graph Search product. In this paper, we describe the data model and query language supported by Unicorn. We also describe its evolution as it became the primary backend for Facebook's search offerings.

1. INTRODUCTION

Over the past three years we have built and deployed a search system called Unicorn[1]. Unicorn was designed with the goal of being able to quickly and scalably search all basic structured information on the social graph and to perform complex set operations on the results. Unicorn resembles traditional search indexing systems [14, 21, 22] and serves its index from memory, but it differs in significant ways because it was built to support social graph retrieval and social ranking from its inception. Unicorn helps products to splice interesting views of the social graph online for new user experiences.

Unicorn is the primary backend system for Facebook Graph Search and is designed to serve billions of queries per day with response latencies less than a few hundred milliseconds. As the product has grown and organically added more features, Unicorn has been modified to suit the product's requirements. This paper is intended to serve as both a narrative of the evolution of Unicorn's architecture, as well as documentation for the major features and components of the system.

To the best of our knowledge, no other online graph retrieval system has ever been built with the scale of Unicorn in terms of both data volume and query volume. The system serves tens of billions of nodes and trillions of edges at scale while accounting for per-edge privacy, and it must also support realtime updates for all edges and nodes while serving billions of daily queries at low latencies.

This paper includes three main contributions:

- We describe how we applied common information retrieval architectural concepts to the domain of the social graph.
- We discuss key features for promoting socially relevant search results.
- We discuss two operators, *apply* and *extract*, which allow rich semantic graph queries.

This paper is divided into four major parts. In Sections 2–5, we discuss the motivation for building unicorn, its design, and basic API. In Section 6, we describe how Unicorn was adapted to serve as the backend for Facebook's *typeahead* search. We also discuss how to promote and rank socially relevant results. In Sections 7–8, we build on the implementation of typeahead to construct a new kind of search engine. By performing multi-stage queries that traverse a series of edges, the system is able to return complex, user-customized views of the social graph. Finally, in Sections 8–10, we talk about privacy, scaling, and the system's performance characteristics for typical queries.

2. THE SOCIAL GRAPH

Facebook maintains a database of the inter-relationships between the people and things in the real world, which it calls the social graph. Like any other directed graph, it consists of *nodes* signifying people and things; and *edges* representing a relationship between two nodes. In the remainder of this paper, we will use the terms *node* and *entity* interchangeably.

Facebook's primary storage and production serving architectures are described in [30]. Entities can be fetched by their primary key, which is a 64-bit identifier (`id`). We also store the edges between entities. Some edges are directional while others are symmetric, and there are many thousands of edge-types. The most well known edge-type

[1] The name was chosen because engineers joked that—much like the mythical quadruped—this system would solve all of our problems and heal our woes if only it existed.

Proceedings of the VLDB Endowment, Vol. 6, No. 11
Copyright 2013 VLDB Endowment 2150-8097/13/09... $ 10.00.

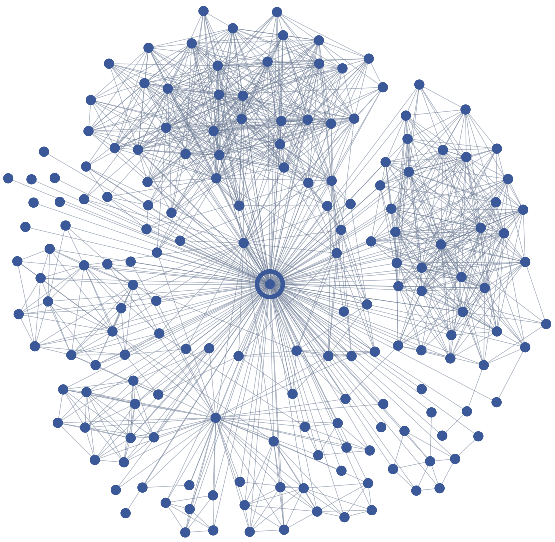


Figure 1: Friend relationships in the social graph. Each friend of the center user is represented by a shaded dot, and each friend relationship among this user's friends is represented by a line.

in the social graph is the `friend` relation, which is symmetric. Another explicit relationship in Facebook's social graph is `likes`, which is an edge from a user to a page that the user has liked. It is common for certain edge-types to be tightly coupled with an inverse edge. The inverse of `likes` is `likers`, which is an edge from a page to a user who has liked that page. There are many other edge-types that are used to build product features. Table 1 lists some of the most common edge-types in the social graph.

Although there are many billions of nodes in the social graph, it is quite sparse: a typical node will have less than one thousand edges connecting it to other nodes. The average user has approximately 130 friends. The most popular pages and applications have tens of millions of edges, but these pages represent a tiny fraction of the total number of entities in the graph.

3. DATA MODEL

Because the social graph is sparse, it is logical to represent it as a set of adjacency lists. Unicorn is an inverted index service that implements an adjacency list service. Each adjacency list contains a sorted list of hits, which are (*DocId*, *HitData*) pairs. We use the words *hit* and *result* interchangeably. A DocId (document identifier) is a pair of (sort-key, `id`), and *HitData* is just an array of bytes that store application-specific data. The sort-key is an integer, and hits are sorted first by sort-key (highest first) and secondly by `id` (lowest first). The sort-key enables us to store the most globally important `id`s earlier in the adjacency list. If an `id` is associated with a sort-key in any adjacency list, it must be associated with the same sort-key in all adjacency lists. We also refer to these adjacency lists as *posting lists*.

In a full-text search system, the HitData would be the place where positional information for matching documents is kept, so our usage of the phrase is somewhat idiosyncratic.

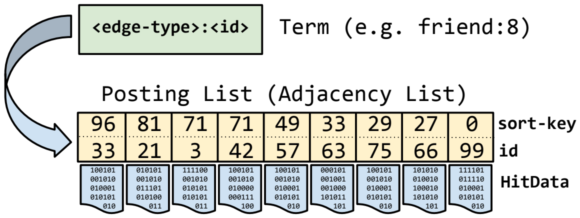


Figure 2: Converting a node's edges into posting lists in an inverted index. Users who are friends of `id` 8 correspond to `ids` in hits of the posting list. Each hit also has a corresponding sort-key and (optional) HitData byte array. Assuming we have three shards and the partitioning function is a simple modulus, we are only showing shard 0 here. There would be similar lists (with different `ids` and sort-keys) for `friend:8` in the other two shards.

$$\begin{array}{lcc} PostingList_n & \rightarrow & (Hit_{n,0}, Hit_{n,1}, ..., Hit_{n,k-1}) \\ Hit_{i,j} & \rightarrow & (DocId_{i,j}, HitData_{i,j}) \\ DocId_{i,j} & \rightarrow & (sort\text{-}key_{i,j}, \mathtt{id}_{i,j}) \end{array}$$

HitData is not present in all terms, as much of the per-`id` data is stored in a separate data structure (see Section 6.2). One application of HitData is for storing extra data useful for filtering results. For example, a posting list might contain the `ids` of all users who graduated from a specific university. The HitData could store the graduation year and major.

Unicorn is *sharded* (partitioned) by result-`id` (the `ids` in query output; as opposed to sharding by input terms) and optimized for handling graph queries. We chose to partition by result-`id` because we wanted the system to remain available in the event of a dead machine or network partition. When queried for friends of Jon Jones, it is better to return some fraction of the friends of Jon Jones than no friends at all. A further advantage of document-sharding instead of term-sharding is that most set operations can be done at the index server (leaf) level instead of being performed higher in the execution stack. This allows us to split the computational load among a greater number of machines, and it also cuts down on bandwidth between machines.

As in most search systems, posting lists are referenced by terms, which, by convention, are of the form:

```
<edge-type>:<id>
```

There is no strict requirement for terms to be of this form, but such terms can be used in special graph operators described in Section 7.1. Edge-type is merely a string such as `friend` or `like`. For example, assume the `id` of user Jon Jones is 5. Clients can request the friend-list for Jon Jones by requesting the term `friend:5`. Figure 2 gives another example of this model.

We can model other real-world structures as edges in Unicorn's graph. For example, we can imagine that there is a node representing the concept *female*, and we can connect all female users to this node via a `gender` edge. Assuming the identifier for this female node is 1, then we can find all friends of Jon Jones who are female by intersecting the sets of result-`ids` returned by `friend:5` and `gender:1`.

Edge-Type	#-out	in-`id`-type	out-`id`-type	Description
`friend`	*hundreds*	USER	USER	Two users are friends (symmetric)
`likes`	*a few*	USER	PAGE	pages (movies, businesses, cities, etc.) liked by a user
`likers`	10–10M	PAGE	USER	users who have liked a page
`live-in`	1000–10M	PAGE	USER	users who live in a city
`page-in`	*thousands*	PAGE	PAGE	pages for businesses that are based in a city
`tagged`	*hundreds*	USER	PHOTO	photos in which a user is tagged
`tagged-in`	*a few*	PHOTO	USER	users tagged in a photo (see Section 7.2)
`attended`	*a few*	USER	PAGE	schools and universities a user attended

Table 1: A sampling of some of the most common edge-types in the social graph. The second column gives the typical number of hits of the output type that will be yielded per term.

4. API AND QUERY LANGUAGE

Clients send queries to Unicorn as Thrift [2, 26] requests, the essential part of which is the *query string*. The client passes the Thrift request to a library that handles sending the query to the correct Unicorn cluster. Internally, client queries are sent to a Unicorn cluster that is in close geographical proximity to the client if possible. Query strings are s-expressions that are composed of several operators (see Table 3), and they describe the set of results the client wishes to receive.

Like many other search systems, Unicorn supports AND and OR operators, which yield the intersection and union of N posting lists, respectively. From the discussion above, a client who wishes to find all female friends of Jon Jones would issue the query `(and friend:5 gender:1)`. If there exists another user Lea Lin with `id` 6, we could find all friends of Jon Jones or Lea Lin by issuing the query `(or friend:5 friend:6)`.

Unicorn also supports a DIFFERENCE operator, which returns results from the first operand that are not present in the second operand. Continuing with the example above, we could find female friends of Jon Jones who are not friends of Lea Lin by using `(difference (and friend:5 gender:1) friend:6)`. As demonstrated by the query above, Unicorn operators support composition, which allows us construct complex queries to create interesting views of the social graph.

For some queries, results are simply returned in DocId ordering. Another common ranking method is to sort results by the number of terms that matched the result. For the query `(or friend:5 friend:6)`, some of the results will match `friend:5`, some will match `friend:6`, and some will match both. In many applications it is useful to give preference to results that match more terms.

5. ARCHITECTURE

Client queries are sent to a Unicorn top-aggregator, which dispatches the query to one rack-aggregator per rack. These rack-aggregators dispatch the query to all index servers in their respective racks. Figure 3 shows a top-aggregator communicating with a single *tier* (a set of all index partitions). Each index server is responsible for serving and accepting updates for one shard of the index. The rack aggregators and top-aggregator are responsible for combining and truncating results from multiple index shards in a sensible way before the top-aggregator finally sends a response back to the client. Rack-aggregators solve the problem that the available bandwidth to servers within a rack is higher than the bandwidth between servers on different racks. Tiers are typically replicated for increased throughput and redundancy, and a rack-aggregator is part of a particular *replica* for a tier, although it knows how to forward requests to another replica in the event that a shard is dead.

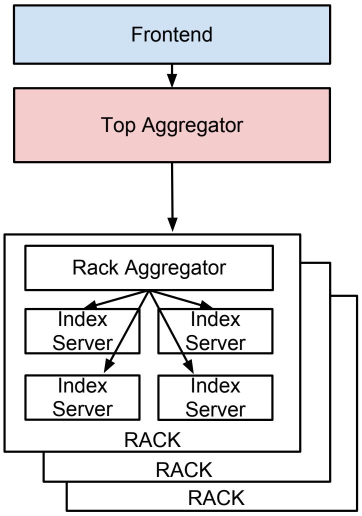


Figure 3: Unicorn tier consisting of three racks. Rack-aggregator processes are stateless and run on every machine (40 or 80) in the racks but are omitted from the diagram for simplicity.

The index server stores adjacency lists and performs set operations on those lists, the results of which are returned to the aggregators. In the index server, intersection, union, and difference operations proceed as is described in standard information retrieval texts [22]. One salient property of our system is that each index shard typically contains a few billion terms, and the total number of unique terms in an entire index of users is typically in the tens of billions. This number increases as more types of entities are added. Although the number of terms is greater than is typical for a full-text search system, the posting lists are generally shorter, with the 50th and 99th percentiles within a representative shard being 3 and 59 hits, respectively. The longest posting list within a shard is typically in the tens of millions of hits. We have experimented with several inverted index formats [17, 22, 31]—both immutable and mutable—to optimize for decoding efficiency and memory footprint. Updates are applied to a mutable index layer, which sits on top of the immutable layer and is treated as a set of additions and subtractions to posting lists in the immutable layer. Because decoding speed is dominated by memory access time for very short posting lists, we get the greatest performance gains by optimizing for decoding the posting lists that consume more than a single CPU-cache line.

As mentioned above, all posting lists are *sharded* (partitioned) by result-`id`. Since our `id`s are assigned essentially randomly, this means that each of our shards has approximately the same number of `id`s assigned to it. Because of this sharding scheme, the full list of hits for a given term will likely be divided across multiple—often all—index partitions. The index server serves all data from memory; it never reads from secondary storage in normal operation except at startup.

Building Indices

Unicorn indices are built using the Hadoop framework [1]. The raw data comes from regular scrapes of our databases that contain the relevant table columns. These database scrapes are accessible via Hive [27]. A client who wishes to put new data into Unicorn merely needs to write the small amount of code necessary to convert Hive rows into `(term, sort-key, id, HitData)` tuples and optional per-`id` metadata. We have a custom Hadoop pipeline that converts these tuples into a Unicorn index, which is then copied out to the relevant production servers.

Because many applications within Facebook require their data to be fresh with the latest up-to-the-minute updates to the social graph, we also support realtime updates. These graph mutations are written by our frontend cluster into Scribe [8, 9]. There is a distributed scribe tailer system that processes these updates and sends the corresponding mutation Thrift requests to Unicorn index servers. For a given update-type, which is known as a *category*, each index server keeps track of the timestamp of the latest update it has received for that category. When a new index server becomes available after a network partition or machine replacement, the scribe tailer can query the index server for its latest update timestamps for each category and then send it all missing mutations.

6. TYPEAHEAD

One of the first major applications of Unicorn was to replace the backend for Facebook's *typeahead* search. Typeahead enables Facebook users to find other users by typing the first few characters of the person's name. Users are shown a list of possible matches for the query, and the results dynamically update as users type more characters. As of the end of 2012, this system serves over $1B$ queries per day, and it is a convenient way for users to navigate across Facebook properties and applications[2].

A typeahead query consists of a string, which is a prefix of the name of the individual the user is seeking. For example, if a user is typing in the name of “Jon Jones” the typeahead backend would sequentially receive queries for “J”, “Jo”, “Jon”, “Jon ”, “Jon J”, etc[3]. For each prefix, the backend will return a ranked list of individuals for whom the user might be searching. Some of these individuals will be within the user's explicit circle of friends and networks.

Index servers for typeahead contain posting lists for every name prefix up to a predefined character limit. These posting lists contain the `id`s of users whose first or last name matches the prefix. A simple typeahead implementation would merely map input prefixes to the posting lists for those prefixes and return the resultant `id`s.

6.1 Social Relevance

One problem with the approach described above is that it makes no provision for social relevance: a query for “Jon” would not be likely to select people named “Jon” who are in the user's circle of friends.

A solution would be to use the And operator to select only results that are friends of the user. For example, if Jon Jones is searching for people whose names begin with “Mel”, the Unicorn query might be something like: `(and mel* friend:3)`. This approach is too simple in practice because it ignores results for users who might be relevant but are not friends with the user performing the search. To handle this problem, we could execute two separate queries—the original query and then another query that selects friends-only—but this is expensive. What we actually want is a way to force some fraction of the final results to possess a trait, while not requiring this trait from all results. This is what the WeakAnd operator accomplishes.

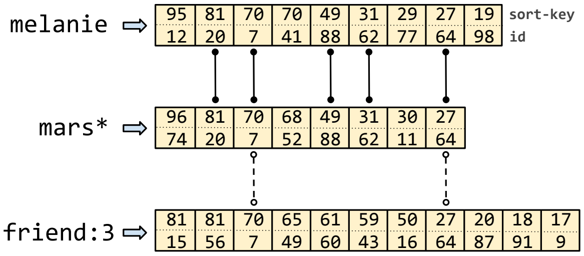


Figure 4: Index structures for query `(weak-and (term friend:3 :optional-hits 2) (term melanie) (term mars*))`. `id`s are shown below sort keys. `id`s returned from this query would be $20, 7, 88,$ and 64. `id` 62 would not be returned because hits 20 and 88 have already exhausted our optional hits.

The WeakAnd operator is a modification of And that allows operands to be missing from some fraction of the results within an index shard. For these *optional terms*, clients can specify an *optional count* or *optional weight* that allows a term to be missing from a certain absolute number or fraction of results, respectively. If the optional count for a term is non-zero , then this term is not required to yield a particular hit, but the index server decrements the optional count for each such non-matching hit. Once the optional count for a particular term reaches zero, the term becomes a *required* term and thus must include the `id` of subsequent hit candidates for them to be included in the result set. Figure 4 shows how WeakAnd can be used when user Jon

[2] The original backend system for typeahead [10] was one of the first large-scale consumer systems to do instantaneous search, and it was already serving hundreds of millions of queries per day. However it was beginning to get difficult to scale as Facebook reached half a billion users, and experimentation was difficult.

[3] We will ignore how the system handles alternative spellings and typos, although there are modules that handle this.

Jones performs the query "Melanie Mars" (a prefix of the full name "Melanie Marshall"). Here, we only allow a finite number of hits to be non-friends of Jon Jones. This ensures that some results will be friends of Jon Jones if any match the text, but it also ensures that we don't miss very good results who are not friends with Jon Jones.

A similar operator to WeakAnd is StrongOr, which is a modification of Or that requires certain operands to be present in some fraction of the matches. StrongOr is useful for enforcing diversity in a result set. For example, if most of Jon Jones' friends live in either Beijing (100), San Francisco (101), or Palo Alto (102), we could fetch a geographically diverse sample of Jon's friends by executing:

```
(strong-or friend:5
     (and friend:5 live-in:100
      :optional-weight 0.2)
     (and friend:5 live-in:101
      :optional-weight 0.2)
     (and friend:5 live-in:102
      :optional-weight 0.1))
```

The optional weights in the operands above mean at least 20% of the results must be people who live in Beijing. Another 20% must be people who live in San Francisco, and 10% must be people who live in Palo Alto. The remainder of results can be people from anywhere (including the cities above). See Table 3 for more information.

6.2 Scoring

It is often useful to return results in an order different from sorting by DocId. For example, we might want to prioritize results for individuals who are close in age to the user typing the query. This requires that we store the age (or birth date) of users with the index. For storing per-entity metadata, Unicorn provides a *forward index*, which is simply a map of `id` to a blob that contains metadata for the id. The forward index for an index shard only contains entries for the `id`s that reside on that shard. Based on Thrift parameters included with the client's request, the client can select a scoring function in the index server that scores each result. The scoring function computes a floating point score for a DocId. It takes several factors as input including the original query, the terms that yielded the DocId, the HitData (if any), and the forward index blob for the `id` (if any). The number of results to score per shard can be specified by the client, but it defaults to the number of results requested multiplied by a constant factor.

The index server returns the results with the highest scores, and aggregators generally give priority to documents with higher scores. However, without some way of maintaining result diversity in the rack-aggregator and top-aggregator, the interesting properties of results yielded by WeakAnd and StrongOr could be lost as results are aggregated and truncated. We have implemented special heap-based data structures for ensuring that results are not strictly selected based on highest scores, but rather that results that add to diversity of the result-set—even if they have lower scores—are given an opportunity to be part of the final result set. More discussion of this scheme can be found in [11].

6.3 Additional Entities

Typeahead can also be used to search for other entity-types in addition to people. Typeahead searches can yield results that are pages, events, and Facebook applications. To support these distinct types of entities while still keeping the size of our tiers at a manageable level, we split Unicorn into multiple entity-type specific tiers—or *verticals*—and modified the top-aggregator to query and combine results from multiple verticals. Edges of the same result-type are placed in the same vertical, so the posting lists for `friend` edges and `likers` edges reside in the same vertical since they both yield user `id`s. This means that set operations like And and Or can still happen at the leaf level, and `id`s which will never be merged or intersected need not be stored together. An alternative implementation would have been to squash all result-types into the same tier, but this led to oversized tiers of many thousands of machines. By partitioning `id`s by vertical, verticals can be replicated as necessary to serve the query load specific to that entity type. An additional advantage of splitting our architecture into verticals is that each vertical can have a team dedicated to managing the data within the tier and building optimal ranking algorithms.

The top-aggregator continues to be the interface through which clients speak to Unicorn. It accepts queries from the frontend and dispatches the query to the necessary verticals. Verticals can be omitted based on properties of the user's query, so different verticals have different load characteristics. Results from verticals are collected, combined, and truncated as necessary to deliver high quality results to the user.

Figure 5 shows an example Unicorn system that has four verticals. We omit vertical replicas from the diagram, although these are typically used to enable higher throughput and fault tolerance. Each vertical has its own unique number of machines, which is indicated in the figure by some verticals being composed of one, two, or three racks. The verticals listed in the figure are for illustrative purposes only, and there are additional verticals in the real production system.

7. GRAPH SEARCH

Many search systems exist which accept text tokens and yield DocIds of some form. However, in a graph search system, there are interesting graph results that are more than one edge away from the source nodes. Without some form of denormalization, this requires supporting queries that take more than one round-trip between the index server and the top-aggregator. Because terms themselves are a combination of an edge-type and an `id`, we can take a pre-specified edge-type from the client and combine it with result `id`s from a round of execution to generate and execute a new query. As an example, we might want to know the pages liked by friends of Melanie Marshall (7) who like Emacs (42). We can answer this by first executing the query `(and friend:7 likers:42)`, collecting the results, and creating a new query that produces the union of the pages liked by any of these individuals:

Inner	`(and friend:7 likers:42)` $\rightarrow 5, 6$
Outer	`(or likes:5 likes:6)`

The ability to use the results of previous executions as seeds for future executions creates new applications for a search system, and this was the inspiration for Facebook's Graph Search consumer product [4]. We wanted to build a general-purpose, online system for users to find entities

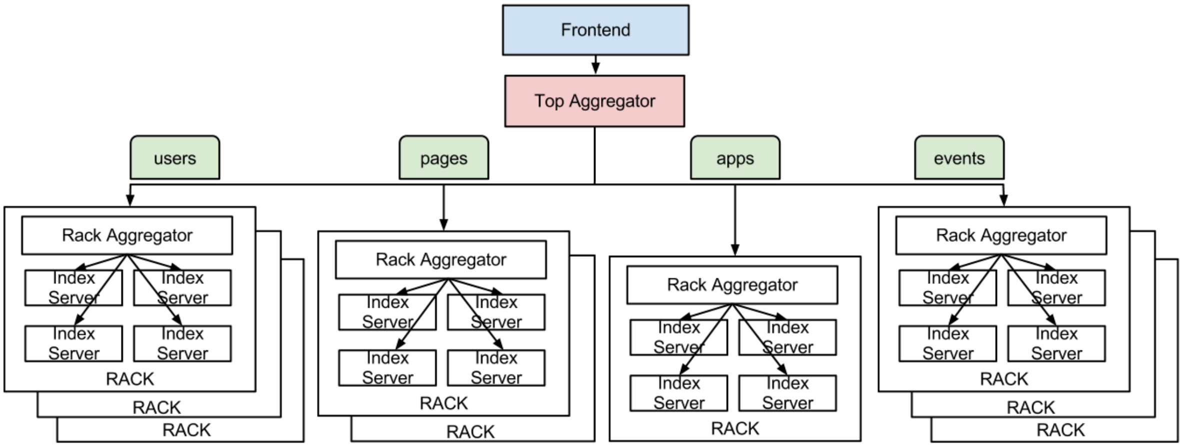


Figure 5: Example of Unicorn cluster architecture with multiple *verticals*. The top-aggregator determines which vertical(s) each query needs to be sent to, and it sends the query to the racks for that vertical. Per-vertical replicas are omitted for simplicity.

in the social graph that matched a set of user-defined constraints. In the following sections, we will describe the additional features we built for Graph Search.

7.1 Apply

A unique feature of unicorn compared to other IR systems is its Apply, or graph-traversal, operator. Apply allows clients to query for a set of `ids` and then use those `ids` to construct and execute a new query. Similar to the *apply* construct in languages like JavaScript, the apply operator is used to perform an operation—in this case prepending a prefix to inner result `ids` and finding the union of the resultant terms—on a set of *inner* query results. Algorithm 1 describes the operator in detail.

Algorithm 1 Apply Operator (in top-aggregator)

```
procedure Apply($prefix, innerQ$)
    $innerResults \leftarrow$ Search($innerQ$)
    $outerQ \leftarrow$ "($or$"
    for each $result$ in $innerResults$ limit $L$ do
        $term \leftarrow$ Concat($prefix$, " : ", $result.id$)
        $outerQ \leftarrow$ Concat($outerQ$, " ", $term$)
    end for
    $outerQ \leftarrow$ Concat($outerQ$, ")")
    $outerResults \leftarrow$ Search($outerQ$)
    return $outerResults$
end procedure
procedure Search($query$)
        ▷ Fetches query results from leaves and merges
    return $results$
end procedure
```

Graph Search adds a photos vertical that tracks hundreds of billions of photo `ids`[4]. The unicorn photos vertical stores a `tagged` edge-type that returns the photo `ids` in which a user is tagged. If a client wants to find all photos of Jon Jones' friends, the client could naively issue a query for `friend:5`, collect the N results, and issue a second query for

`(or tagged:<`id_0`> ...  tagged:<`id_{n-1}`>)`

However, the apply operator allows the client to retrieve photos-of-friends in one query and push most of the computational cost down to the backend:

```
(apply tagged: friend:5)
```

In an Apply operation, the latter operand (`friend:5` in Figure 6) is executed first as the so-called *inner* query (first round), and these inner results are prepended with the first operand (`tagged:`) and conjoined with an Or for the *outer* query (second round). As shown in Algorithm 1, we limit the number of generated operands for the outer query to some limit L. This limit can be overridden by clients but is typically around 5000 for performance reasons.

When the client uses Apply, Unicorn is essentially doing what the client would have done. However, the network latency saved by doing extra processing within the Unicorn cluster itself and the ability write the aggregation and processing of the extra steps in a high performance language makes Apply a good "win". In addition, client code is simplified because clients do not need to deal with messy aggregation and term-generation code for multiple passes. In some sense, Apply is merely *syntactic sugar* to allow the system to perform expensive operations lower in the hardware stack. However, by allowing clients to show semantic intent, further optimizations are sometimes possible (see query planning discussion in Section 7.1.2 below). This scheme also allows more automation related to training collaborative inner-outer query ranking models. It is helpful to think of Apply as analogous to Join because it can leverage aggregate data and metadata to execute queries more efficiently than clients themselves.

7.1.1 Example: Friends-of-Friends

The canonical example for apply is finding so-called *friends-of-friends* of a user. Jon Jones's friends-of-friends are depicted in figure 6. Jon's friends are Amy, Bob, Chuck, Dana, and Ed. Jon's friends-of-friends are simply the friends

[4] Users can "tag" their friends in photos to create a structured assocation between the individual and the photo, and photos typically have zero or a few tags.

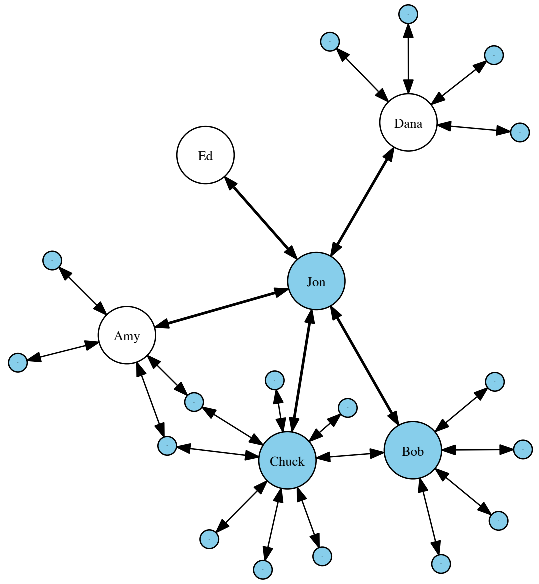


Figure 6: Friends-of-friends for user Jon Jones. All shaded circles would be returned.

of these individuals. Because Chuck and Bob are mutual friends, they are also friends-of-friends of Jon. Jon is also a friend-of-friends of himself since he is—by definition—a friend of his friends. Clients can use the Difference operator to eliminate Jon and his friends from the friend-of-friends result-set if they desire. Not all of Jon's friends are contained in the set of his friends-of-friends, so clients often take the union of these sets with the Or operator.

The top-aggregator handles processing the client's request in multiple passes. In the first pass of a query such as `(apply friend: friend:5)`, the top-aggregator sends the inner query to all *users* racks first. The top-aggregator has logic to map term prefixes to the appropriate vertical, or clients can manually specify a vertical for the inner and outer query. Once it collects the results for the inner query, it constructs an "outer" query based on the results of the inner query. In our example, if the results of the inner query are `ids`: 10,11,13,17,18 then the top-aggregator would construct a new query `(or friend:10 friend:11 friend:13 friend:17 friend:18)` and send this new query, again, to all *users* racks. Once the results for this outer query are gathered, the results are re-sorted and truncated if necessary, and then they are returned to the client.

We support arbitrary nesting of apply operators. Because of the latency associated with adding an extra round of backend fetching and the CPU-cost and bandwidth of having more than a few hundred terms in a query, most apply operations are one level. We have found that friends-of-friends is very useful in practice, but friends-of-friends-of-friends adds virtually no benefit. However we have clients who regularly employ multi-level apply operations.

Apply allows us to serve interesting views on the social graph on-line without being required to denormalize the data. Considering the friends-of-friends example again, there are several possible ways such a query could be implemented. Naively, we could explicitly store the list of friends-of-friends for every user and reference it via

	friend: only	**F-o-F inline**
Est. #Hits/User	130	48k
Est. Total Bytes	484GB	178TB
Avg. Latency	20ms	7ms
Variance	medium	low

Table 2: Comparing friends-of-friends (F-o-F) implementations

```
friend-of-friend:<user>
```

. However such an implementation would take up an unreasonable amount of space: While each user has 130 friends on average, they have approximately 48000 friends-of-friends[5]. Even assuming only 4 bytes per posting list entry, this would still take over 178TB to store the posting lists for 1 billion users, which would require hundreds of machines.

Compare such an approach to our `apply`-based implementation in Unicorn. Here, we only have terms for `friend:`. Given the same assumptions, this corpus would take up 484GB and would fit comfortably on a handful of machines. In most situations, the trade-off in extra space one makes by inlining friends-of-friends is not worth the execution time saved by an extra round-trip to the index server. The denormalization approach also suffers from being necessarily ad hoc. It can only cover the limited range of query patterns that are anticipated at indexing time. Another approach for finding friend-of-friends is discussed in [29].

The Apply operator is often used with *term-count* ranking. Instead of merely returning Jon Jones's friends-of-friends ordered by DocId, we can rank them by the number of terms that matched each result. In figure 6, we can notice that some of the colored circles have more edges than others. These edges represent how many common friends these individuals have with Jon Jones, and the edge count is a good proxy for social distance.

7.1.2 Example: Restaurants

Assume we wanted to answer the query: "Restaurants in San Francisco liked by people from Beijing", and assume our index has a term `page-type:restaurant` that returns pages that are restaurants. To answer the query, we can attempt to use the following s-expression:

```
(and (term page-type:restaurant)
     (term place-in:102)
     (apply likes: (term live-in:100)))
```

The inner query of the Apply operator would have millions of results, which is too many to return to the top-aggregator — let alone to use for constructing an outer query. In practice, the highest number of terms we can use as part of the outer query is around 10^5 because of network and CPU latency issues[6]. Hence we are forced to truncate the inner results, which may lead to perfectly good restaurants in San

[5] This number differs from the expected quantity of $130^2 = 16900$ because a user's friends typically have more friends than the global average.

[6] Specifically, each query term lookup typically causes at least one CPU-cache miss. Ignoring thread parallelism and assuming 100ns per main memory reference, this will already take $100000 * 100\,\text{ns} = 10\,\text{ms}$, which is a significant fraction of our per-index-server request timeout.

Operator	Example S-expression	Description
term	`(term friend:5)`	Friends of Jon Jones (Terms may also appear naked inside of other operands.)
and	`(and friend:5 friend:6)`	Friends of Jon Jones who are also friends of Lea Lin
or	`(or friend:5 friend:6 friend:7)`	Friends of Jon Jones or Lea Lin or Melanie Marshall
difference	`(difference friend:5 (or friend:6 friend:7))`	Friends of Jon Jones who are not friends of Lea Lin or Melanie Marshall

weak-and	`(weak-and friend:6`
	`  (term friend:7 :optional-weight 0.2))`
		Friends of Lea Lin, of whom at least 80% must also be friends of Melanie Marshall. Number of non-matching hits can be specified as an absolute number or as a fraction of total results (see Section 6.1)

strong-or	`(strong-or`
	`  (term live-in:101 :optional-weight 0.1)`
	`  (term live-in:102 :optional-weight 0.1))`
		People who live in Palo Alto or San Francisco. At least 10% of results must be San Francisco residents, and 10% must be Palo Alto residents

apply	`(apply friend: (and friend:5 friend:6))`	Friends of Jon and Lea Lin's mutual friends

Table 3: Unicorn operators. Jon Jones is user 5, Lea Lin is 6, and Melanie Marshall is 7. San Francisco and Palo Alto are 101 and 102, respectively.

Francisco being omitted from the result set because of *inner apply truncation*. We will describe three approaches for solving this problem.

First, we can be smarter about which results we return from the inner query. For example, at scoring-time, we could prioritize users who have ever lived or worked near San Francisco, since they are more likely to have liked a restaurant in San Francisco.

Second, we can selectively denormalize edges if a use-case is common enough. Here, we could create a new edge, `liked-page-in`, that returns the set of users who have ever liked a page associated with a particular city (this data-set can be computed offline). Given such an edge, we can convert the inner part of the Apply to:

```
(and (term live-in:100)
     (term liked-page-in:101))
```

This ensures that we select only users who have liked pages in San Francisco. Another viable, albeit memory intensive, approach would be to fully denormalize away the Apply altogether. We could create a new edge, `in-X-likefrom:Y`, that returns pages in some location, X, liked by people from another location, Y[7]. If all such edges were indexed, we could answer the original user query by executing:

```
(and
  (term
   pages-in-101-liked-by-people-from:100)
  (term page-type:restaurant))
```

Another downside of this approach is that it might be interesting to know *which* users from Beijing like the returned San Francisco restaurants, although this could be solved by performing additional queries to select these users.

A final strategy for solving the inner apply truncation problem is to use query planning to avoid inner truncation altogether. This is analogous to query planning in a DBMS for complicated JOIN operations [23, 25]. For our original query, the top-aggregator could recognize that there are fewer restaurants in San Francisco than there are users from Beijing. Thus, we could first execute a query to find all restaurants in San Francisco, and then we could find all likers of these restaurants who live in Beijing. This requires that we work backwards from the final results (people from Beijing who like restaurants in San Francisco) to get the desired results (the restaurants themselves), but this is straightforward assuming we keep track of which terms from the outer query matched our final results.

A way to directly express this approach in the query language, which comes at the cost of an extra round of backend fetches, is to intersect the inner query itself with an apply that effectively *inverts* the original query. In this approach, the modified query would be:

```
(and (term page-type:restaurant)
     (term place-in:102)
     (apply likes:
            (and live-in:100
                 (apply likers:
                        (and
                        (term page-in:101)
                        (term page-type:restaurant)
                                                 )))))
```

In practice, good inner result ranking solves the vast majority of truncation problems, although this is an area of active monitoring and development. Initial deployments of

[7] There is no constraint in our index implementation that prevents terms from being parameterized by two `id`s, although space would be a concern. One way to prevent the index size from blowing up would be to exclude terms with fewer than K hits.

selective denormalization and *apply inversion* show promising results.

7.2 Extract

The EXTRACT operator was created as a way to tackle a common use-case: Say a client wishes to look up "People tagged in photos of Jon Jones". Given the architecture we have discussed so far, the natural solution would be to use the APPLY operator to look up photos of Jon Jones in the photos vertical and then to query the users vertical for people tagged in these photos. However, this approach blows up our index size because it creates hundreds of billions of new terms in the users vertical. To merely store pointers to the posting lists for these terms—not term hashes or the terms themselves—it would require hundreds of GB *per shard*, which is more than the amount of RAM in a single machine.

For this case of billions of "one-to-few" mappings, it makes better sense to store the result `id`s in the forward index of the secondary vertical and do the lookup inline. For the example above, this means that we store the `id`s of people tagged in a photo in the forward index data for that photo in the photos vertical. This is a case where partially denormalizing the data actually saves space, and this inline lookup of desired `id`s from the forward index is what the EXTRACT operator does.

Algorithm 2 describes how extract works on the index server. For the sake of simplicity, we ignore scoring. In some cases it makes sense to score (and possibly reject) results from `iter` before looking up the extracted result. Sometimes the extracted results are sent to their natural vertical to be scored and ranked. We also omit the implementation of the `ExtractSet`, although this is non-trivial. A disadvantage of this approach is that the same extracted result may be returned by multiple shards, leaving duplicate removal as a task for the top-aggregator. Because extract does not fundamentally require extra round-trips—unlike the apply operator—performance overhead of extract is minimal. The extra forward index lookup adds very little extra computational cost, since, typically, the forward index entry has already been fetched in order to score the source document.

Algorithm 2 EXTRACT Operator (in index server)

```
procedure EXTRACT($query, extractKey$)
    $extractSet \leftarrow None$
    $eKey \leftarrow extractKey$
    $iter = execute(query)$
    for each $result$ in $iter$ do
        $e = $ LOOKUP$(fwdIndex, result.id, eKey)$
        INSERT$(extractSet, e)$
    end for
    return $extractSet.results$
end procedure
```

8. LINEAGE

Facebook takes privacy very seriously, and it is important not to show results that would violate the privacy settings of our users. Certain graph edges cannot be shown to all users but rather only to users who are friends with or in the same network as a particular person.

Unicorn itself does not have privacy information incorporated into its index. Instead, our approach is to give callers all the relevant data concerning how a particular result was generated in Unicorn so that the caller—typically our PHP frontend—can make a proper privacy check on the result.

This design decision imposes a modest efficiency penalty on the overall system, but it has two major justifications. First, we did not want our system to provide the strict consistency and durability guarantees that would be needed for a full privacy solution. If a user "un-friends" another user, the first user's friends-only content immediately and irrevocably must become invisible to the second user. Even a 30-second lag-time is unacceptable. The engineering effort required to guarantee that our realtime pipeline stayed strictly up-to-date was deemed overly burdensome. Dead and flaky machines add further complications. In *CAP*-speak, we chose availability and partition tolerance. Unicorn is not an authoritative database system.

An even stronger reason to keep privacy logic separate from Unicorn is the DRY ("Don't Repeat Yourself") principle of software development [19]. Facebook has organically accumulated a massive set of complex, privacy-related business logic in its frontend. Good software design dictates that we leverage this logic instead of replicating it in another place. The modest efficiency costs of this approach only get progressively lower as Facebook invests in broad efficiency initiatives [3, 6].

To enable clients to make privacy decisions, a string of metadata is attached to each search result to describe its *lineage*. Lineage is a structured representation of the edges that were traversed in order to yield a result. Figure 7 gives an example of what is represented by the lineage for a particular search result. The lineage is serialized as JSON or a Thrift structure and added to each result in the response Thrift object.

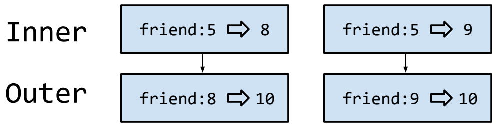


Figure 7: *Lineage* for Chuck (10) in the query represented by Figure 6. Amy (8) is friends with Jon Jones (5), and Chuck is friends with Amy. Alternatively, Bob (9) is friends with Jon Jones, and Chuck is also friends with Bob. For the user to be able to see that Chuck is a friend-of-friends of Jon Jones, the user must be allowed to see either the two edges on the LHS or the two edges on the RHS (or all four edges).

Given the lineage for an `id` in the Unicorn results, our frontend can query an authoritative privacy service to determine if sufficient edges are visible to include the `id` in the user's search results. While the lineage for a single result can sometimes take many kilobytes, the bandwidth and computational cost for processing lineage has not been a major hindrance for Graph Search, as Facebook infrastructure is already designed to spend significant resources on privacy checks.

9. HANDLING FAILURES AND SCALE

A system with thousands of machines in a single replica is bound to experience frequent machine failures and occasional network partitions. Sharding by `id` provides some degree of "passive" safety, since serving incomplete results for a term is strongly preferable to serving empty results. As is typical of many cluster architectures, Facebook has systems for automatically replacing dead machines, so replacements can often be spun-up within a small multiplier of the time it takes to copy the needed index data to the new machine.

Additionally, having multiple copies of a vertical provides for both horizontal scalability and redundancy. Rack aggregators know how to forward requests to a shard copy in another rack in the event that a shard becomes unavailable. Several checks are performed to ensure that such redundant requests are not executed during peak system load or when cascading failures might be possible.

Each vertical can have a different replication factor to optimize for the load and redundancy requirements specific to that vertical. Additionally, analysis sometimes shows that particular term families within a vertical are especially "hot". In these cases, it is possible to build a custom, smaller subset of a particular vertical to provide extra replication for these hot term families. For instance, analysis showed that `friend:` and `likers:` terms were especially common. We created a special "friends and fans" tier-type with a high replication factor, and we added logic to the top-aggregator to detect these queries and forward them to the new tiers. This enabled us to reduce the replication factor of the users vertical and realize significant capital savings.

10. PERFORMANCE EVALUATION

We have executed some benchmarks to provide a better understanding of basic query performance and APPLY performance. All queries below were executed on a set of seven racks consisting of 37 identical machines per rack (259 shards). Each machine is built per Facebook's Open Compute [7] spec and has dual 2.2GHz Sandy Bridge Intel CPUs and a 10 Gbps NIC. The index contains `friend:` and `likers:` terms and fits in memory.

As a baseline, we evaluated the query "People who like Computer Science", which corresponds to the s-expression:

```
(term likers:104076956295773)
```

There are over 6M likers of computer science on Facebook. For all queries, we requested 100 results. Table 4 shows the relevant data for the query above.

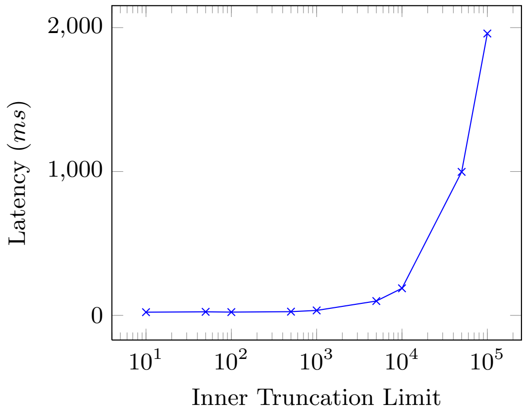
Next, we build an APPLY on top of this base query by finding the friends of these individuals:

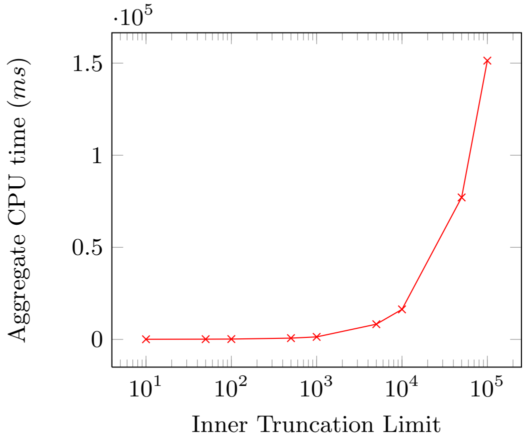
```
(apply friend: likers:104076956295773)
```

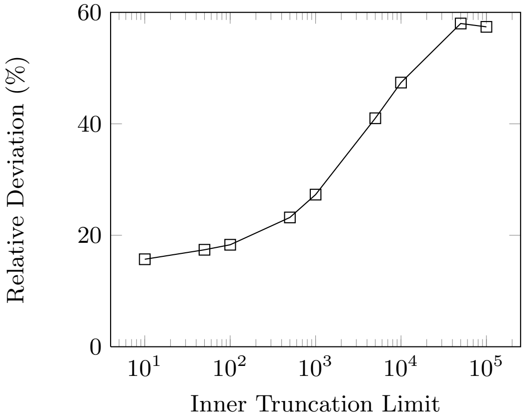
Since there are millions of results for the inner query, the inner result set will be truncated before executing the outer query. The graphs below show how performance is affected as the inner truncation limit is increased from 10 to 100k.

Query	`likers:104076956295773`
Latency	11ms
Aggregate CPU	31220μs
Relative deviation	17.7%

Table 4: Relevant statistics for baseline query "People who like computer science". The query was run 100 times, and 100 results were requested each time. These are the average statistics. Latency is measured in wall-time as seen from a caller to the top-aggregator. Aggregate CPU is the total CPU time (not wall time) spent executing the query summed across all 37 index servers. Relative deviation is computed as the standard deviation of index server response times (as seen by the rack aggregators) normalized by the mean response time.







The first two plots above show that increasing the inner truncation limit leads to higher latency and cost, with latency passing $100ms$ at approximately a limit of 5000. Query cost increases sub-linearly, but there will likely always be queries whose inner result sets will need to be truncated to prevent latency from exceeding a reasonable threshold (say, $100ms$).

This means that—barring architectural changes—we will always have queries that require inner truncation. As mentioned in section 7.1.2, good inner query ranking is typically the most useful tool. However, it is always possible to construct "needle-in-a-haystack" queries that elicit bad performance. Query planning and selective denormalization can help in many of these cases.

The final plot shows that relative deviation increases gradually as the truncation limit increases. Larger outer queries have higher variance per shard as perceived by the rack aggregator. This is likely because network queuing delays become more likely as query size increases.

11. RELATED WORK

In the last few years, sustained progress has been made in scaling graph search via the SPARQL language, and some of these systems focus, like Unicorn, on real-time response to ad-hoc queries [24, 12]. Where Unicorn seeks to handle a finite number of well understood edges and scale to trillions of edges, SPARQL engines intend to handle arbitrary graph structure and complex queries, and scale to tens of millions of edges [24]. That said, it is interesting to note that the current state-of-the-art in performance is based on variants of a structure from [12] in which data is vertically partitioned and stored in a column store. This data structure uses a clustered B+ tree of (subject-id, value) for each property and emphasizes merge-joins, and thus seems to be evolving toward a posting-list-style architecture with fast intersection and union as supported by Unicorn.

Recently, work has been done on adding keyword search to SPARQL-style queries [28, 15], leading to the integration of posting lists retrieval with structured indices. This work is currently at much smaller scale than Unicorn. Starting with XML data graphs, work has been done to search for subgraphs based on keywords (see, e.g. [16, 20]). The focus of this work is returning a *subgraph*, while Unicorn returns an *ordered list of entities.*

In some work [13, 18], the term 'social search' in fact refers to a system that supports question-answering, and the social graph is used to predict which person can answer a question. While some similar ranking features may be used, Unicorn supports queries *about* the social graph rather than *via* the graph, a fundamentally different application.

12. CONCLUSION

In this paper, we have described the evolution of a graph-based indexing system and how we added features to make it useful for a consumer product that receives billions of queries per week. Our main contributions are showing how many information retrieval concepts can be put to work for serving graph queries, and we described a simple yet practical multiple round-trip algorithm for serving even more complex queries where edges are not denormalized and instead must be traversed sequentially.

13. ACKNOWLEDGMENTS

We would like to acknowledge the product engineers who use Unicorn as clients every day, and we also want to thank our production engineers who deal with our bugs and help extinguish our fires. Also, thanks to Cameron Marlow for contributing graphics and Dhruba Borthakur for conversations about Facebook's storage architecture. Several Facebook engineers contributed helpful feedback to this paper including Philip Bohannon and Chaitanya Mishra. Bret Taylor was instrumental in coming up with many of the original ideas for Unicorn and its design. We would also like to acknowledge the following individuals for their contributions to Unicorn: Spencer Ahrens, Neil Blakey-Milner, Udeepta Bordoloi, Chris Bray, Jordan DeLong, Shuai Ding, Jeremy Lilley, Jim Norris, Feng Qian, Nathan Schrenk, Sanjeev Singh, Ryan Stout, Evan Stratford, Scott Straw, and Sherman Ye.

Open Source

All Unicorn index server and aggregator code is written in C++. Unicorn relies extensively on modules in Facebook's "Folly" Open Source Library [5]. As part of the effort of releasing Graph Search, we have open-sourced a C++ implementation of the Elias-Fano index representation [31] as part of Folly.

14. REFERENCES

[1] Apache Hadoop. `http://hadoop.apache.org/`.

[2] Apache Thrift. `http://thrift.apache.org/`.

[3] Description of HHVM (PHP Virtual machine). `https://www.facebook.com/note.php?note_id=10150415177928920`.

[4] Facebook Graph Search. `https://www.facebook.com/about/graphsearch`.

[5] Folly GitHub Repository. `http://github.com/facebook/folly`.

[6] HPHP for PHP GitHub Repository. `http://github.com/facebook/hiphop-php`.

[7] Open Compute Project. `http://www.opencompute.org/`.

[8] Scribe Facebook Blog Post. `http://www.facebook.com/note.php?note_id=32008268919`.

[9] Scribe GitHub Repository. `http://www.github.com/facebook/scribe`.

[10] The Life of a Typeahead Query. `http://www.facebook.com/notes/facebook-engineering/the-life-of-a-typeahead-query/389105248919`.

[11] Under the Hood: Indexing and Ranking in Graph Search. `http://www.facebook.com/notes/facebook-engineering/indexing-and-ranking-in-graph-search/10151361720763920`.

[12] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[13] L. A. Adamic, J. Zhang, E. Bakshy, and M. S. Ackerman. Knowledge sharing and Yahoo Answers: everyone knows something. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 665–674, New York, NY, USA, 2008. ACM.

[14] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar. 2003.

[15] S. Elbassuoni and R. Blanco. Keyword search over RDF graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 237–242, 2011.

[16] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, 2003.

[17] S. Heman. Super-Scalar Database Compression Between RAM and CPU-Cache. In *MS Thesis, Centrum voor Wiskunde en Informatica (CWI)*, 2005.

[18] D. Horowitz and S. D. Kamvar. The anatomy of a large-scale social search engine. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 431–440, 2010.

[19] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 505–516. VLDB Endowment, 2005.

[21] B. Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (2. ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2011.

[22] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[23] T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.

[24] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[25] R. Ramakrishnan and J. Gehrke. *Database Management Systems (3. ed.)*. McGraw-Hill, 2003.

[26] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: scalable cross-language services implementation, 2007. `http://thrift.apache.org/static/files/thrift-20070401.pdf`.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive-a warehousing solution over a map-reduce framework. In *VLDB '09: Proceedings of the VLDB Endowment*, pages 1626–1629, 2009.

[28] A. Tonon, G. Demartini, and P. Cudré-Mauroux. Combining inverted indices and structured search for ad-hoc object retrieval. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 125–134, 2012.

[29] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proc. 6th ACM International Conference on Web Search and Data Mining*. WSDM, 2013.

[30] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 791–792, 2012.

[31] S. Vigna. Quasi-succinct indices. In *Proceedings of the Sixth International Conference on Web Search and Web Data Mining, WSDM 2013*, pages 83–92. ACM, 2013.