The power of the graph

Facebook puts an extremely demanding workload on its data backend. Every time any one of over a billion active users visits Facebook through a desktop browser or on a mobile device, they are presented with hundreds of pieces of information from the social graph. Users see News Feed stories; comments, likes, and shares for those stories; photos and check-ins from their friends -- the list goes on. The high degree of output customization, combined with a high update rate of a typical user's News Feed, makes it impossible to generate the views presented to users ahead of time. Thus, the data set must be retrieved and rendered on the fly in a few hundred milliseconds.

This challenge is made more difficult because the data set is not easily partitionable, and by the tendency of some items, such as photos of celebrities, to have request rates that can spike significantly. Multiply this by the millions of times per second this kind of highly customized data set must be delivered to users, and you have a constantly changing, read-dominated workload that is incredibly challenging to serve efficiently.

Memcache and MySQL

Facebook has always realized that even the best relational database technology available is a poor match for this challenge unless it is supplemented by a large distributed cache that offloads the persistent store. Memcache has played that role since Mark Zuckerberg installed it on Facebook's Apache web servers back in 2005. As efficient as MySQL is at managing data on disk, the assumptions built into the InnoDB buffer pool algorithms don't match the request pattern of serving the social graph. The spatial locality on ordered data sets that a block cache attempts to exploit is not common in Facebook workloads. Instead, what we call creation time locality dominates the workload -- a data item is likely to be accessed if it has been recently created. Another source of mismatch between our workload and the design assumptions of a block cache is the fact that a relatively large percentage of requests are for relations that do not exist -- e.g., "Does this user like that story?" is false for most of the stories in a user's News Feed. Given the overall lack of spatial locality, pulling several kilobytes of data into a block cache to answer such queries just pollutes the cache and contributes to the lower overall hit rate in the block cache of a persistent store.

The use of memcache vastly improved the memory efficiency of caching the social graph and allowed us to scale in a cost-effective way. However, the code that product engineers had to write for storing and retrieving their data became quite complex. Even though memcache has "cache" in its name, it's really a general-purpose networked in-memory data store with a key-value data model. It will not automatically fill itself on a cache miss or maintain cache consistency. Product engineers had to work with two data stores and very different data models: a large cluster of MySQL servers for storing data persistently in relational tables, and an equally large collection of memcache servers for storing and serving flat key-value pairs derived (some indirectly) from the results of SQL queries. Even with most of the common chores encapsulated in a data access library, using the memcache-MySQL combination efficiently as a data store required quite a bit of knowledge of system internals on the part of product engineers. Inevitably, some made mistakes that led to bugs, user-visible inconsistencies, and site performance issues. In addition, changing table schemas as products evolved required coordination between engineers and MySQL cluster operators. This slowed down the change-debug-release cycle and didn't fit well with Facebook's "move fast" development philosophy.

Objects and associations

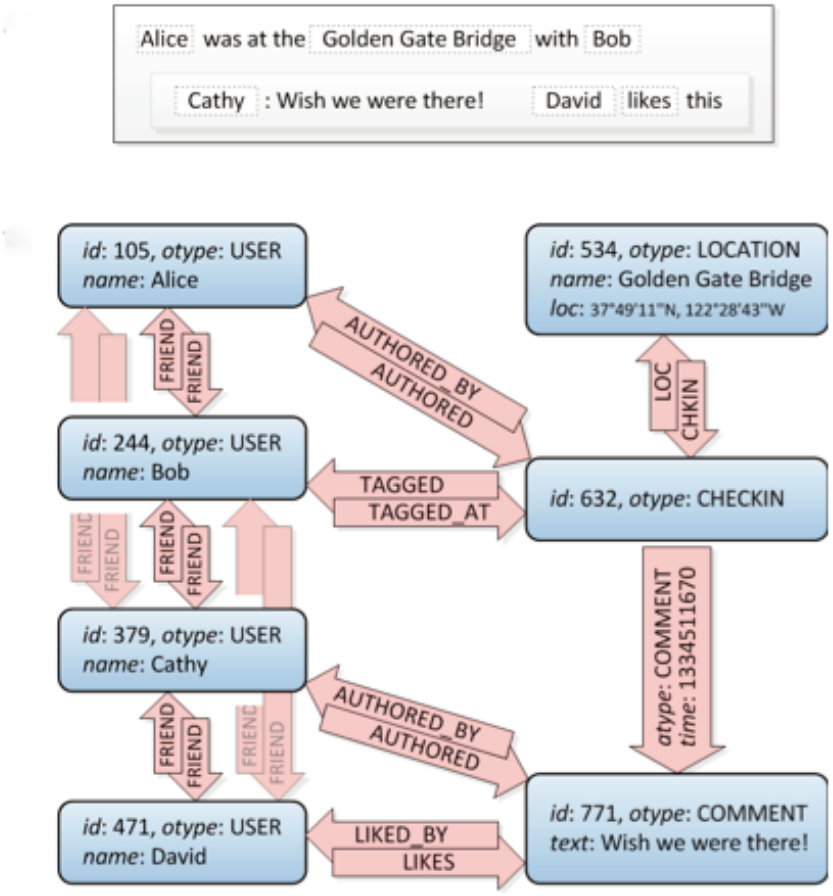
In 2007, a few Facebook engineers set out to define new data storage abstractions that would fit the needs of all but the most demanding features of the site while hiding most of the complexity of the underlying distributed data store from product engineers. The Objects and Associations API that they created was based on the graph data model and was initially implemented in PHP and ran on Facebook's web servers. It represented data items as nodes (objects), and relationships between them as edges (associations). The API was an immediate success,

with several high-profile features, such as likes, pages, and events implemented entirely on objects and associations, with no direct memcache or MySQL calls.

As adoption of the new API grew, several limitations of the client-side implementation became apparent. First, small incremental updates to a list of edges required invalidation of the entire item that stored the list in cache, reducing hit rate. Second, requests operating on a list of edges had to always transfer the entire list from memcache servers over to the web servers, even if the final result contained only a few edges or was empty. This wasted network bandwidth and CPU cycles. Third, cache consistency was difficult to maintain. Finally, avoiding thundering herds in a purely client-side implementation required a form of distributed coordination that was not available for memcache-backed data at the time.

All those problems could be solved directly by writing a custom distributed service designed around objects and associations. In early 2009, a team of Facebook infrastructure engineers started to work on TAO ("The Associations and Objects"). TAO has now been in production for several years. It runs on a large collection of geographically distributed server clusters. TAO serves thousands of data types and handles over a billion read requests and millions of write requests every second. Before we take a look at its design, let's quickly go over the graph data model and the API that TAO implements.

TAO data model and API



This simple example shows a subgraph of objects and associations that is created in TAO after Alice checks in at the Golden Gate Bridge and tags Bob there, while Cathy comments on the check-in and David likes it. Every data item, such as a user, check-in, or comment, is represented by a typed object containing a dictionary of named fields. Relationships between objects, such as "liked by" or "friend of," are represented by typed edges (associations) grouped in association lists by their origin. Multiple associations may connect the same pair of

objects as long as the types of all those associations are distinct. Together objects and associations form a labeled directed multigraph.

For every association type a so-called inverse type can be specified. Whenever an edge of the direct type is created or deleted between objects with unique IDs id1 and id2, TAO will automatically create or delete an edge of the corresponding inverse type in the opposite direction (id2 to id1). The intent is to help the application programmer maintain referential integrity for relationships that are naturally mutual, like friendship, or where support for graph traversal in both directions is performance critical, as for example in “likes” and “liked by."

The set of operations on objects is of the fairly common create / set-fields / get / delete variety. All objects of a given type have the same set of fields. New fields can be registered for an object type at any time and existing fields can be marked deprecated by editing that type’s schema. In most cases product engineers can change the schemas of their types without any operational work.

Associations are created and deleted as individual edges. If the association type has an inverse type defined, an inverse edge is created automatically. The API helps the data store exploit the creation-time locality of workload by requiring every association to have a special time attribute that is commonly used to represent the creation time of association. TAO uses the association time value to optimize the working set in cache and to improve hit rate.

There are three main classes of read operations on associations:

- Point queries look up specific associations identified by their (id1, type, id2) triplets. Most often they are used to check if two objects are connected by an association or not, or to fetch data for an association.

- Range queries find outgoing associations given an (id1, type) pair. Associations are ordered by time, so these queries are commonly used to answer questions like "What are the 50 most recent comments on this piece of content?" Cursor-based iteration is provided as well.

- Count queries give the total number of outgoing associations for an (id1, type) pair. TAO optionally keeps track of counts as association lists grow and shrink, and can report them in constant time

We have kept the TAO API simple on purpose. For instance, it does not offer any operations for complex traversals or pattern matching on the graph. Executing such queries while responding to a user request is almost always a suboptimal design decision. TAO does not offer a server-side set intersection primitive. Instead we provide a client library function. The lack of clustering in the data set virtually guarantees that having the client orchestrate the intersection through a sequence of simple point and range queries on associations will require about the same amount of network bandwidth and processing power as doing such intersections entirely on the server side. The simplicity of TAO API helps product engineers find an optimal division of labor between application servers, data store servers, and the network connecting them.

Implementation

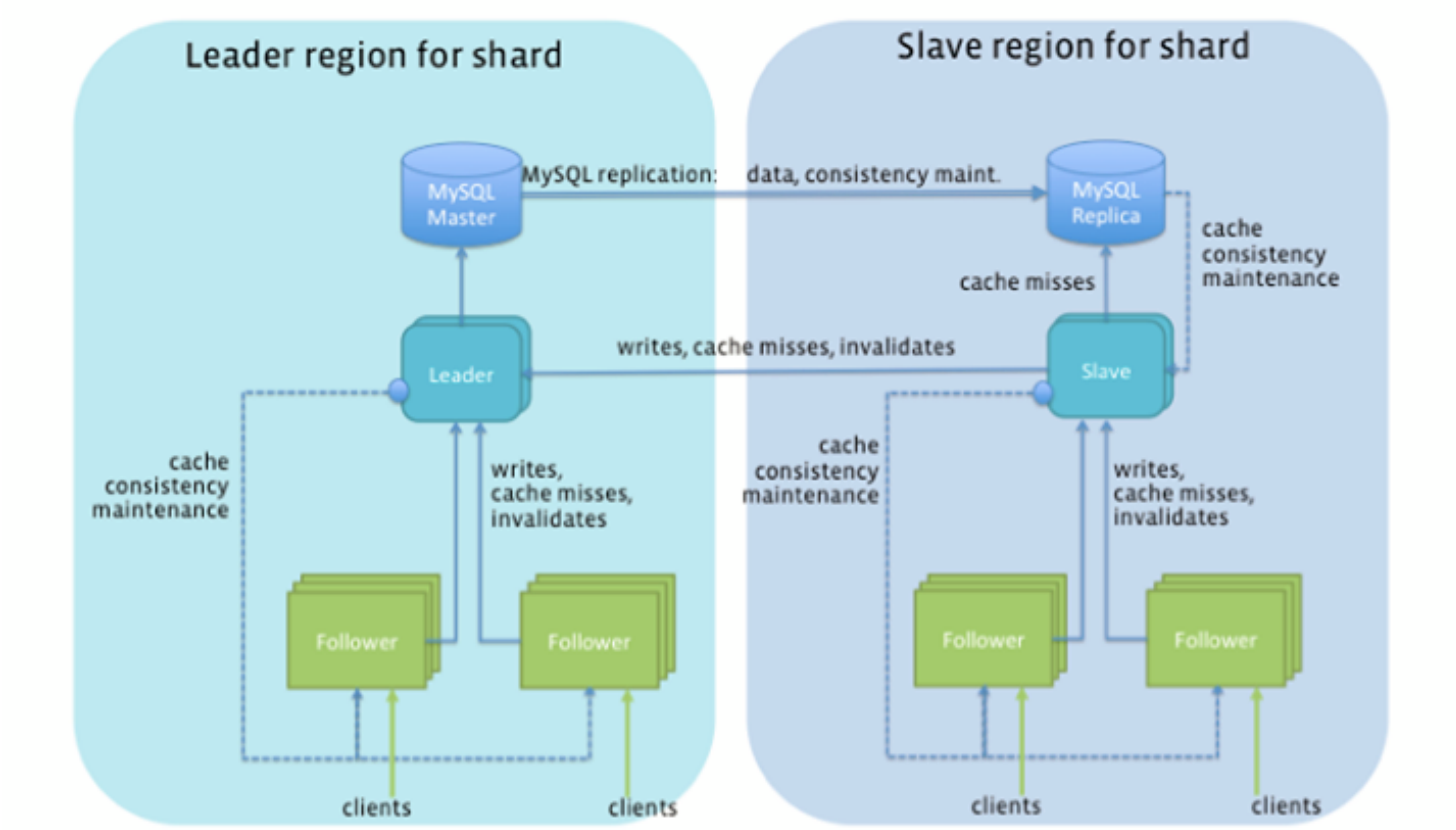
The TAO service runs across a collection of server clusters geographically distributed and organized logically as a tree. Separate clusters are used for storing objects and associations persistently, and for caching them in RAM and Flash memory. This separation allows us to scale different types of clusters independently and to make efficient use of the server hardware.

Client requests are always sent to caching clusters running TAO servers. In addition to satisfying most read requests from a write-through cache, TAO servers orchestrate the execution of writes and maintain cache consistency among all TAO clusters. We continue to use MySQL to manage persistent storage for TAO objects and associations.

The data set managed by TAO is partitioned into hundreds of thousands of shards. All objects and associations

in the same shard are stored persistently in the same MySQL database, and are cached on the same set of servers in each caching cluster. Individual objects and associations can optionally be assigned to specific shards at creation time. Controlling the degree of data collocation proved to be an important optimization technique for reducing communication overhead and avoiding hot spots.

Shards can be migrated or cloned among servers in the same cluster to equalize the load and to smooth out load spikes. Load spikes are common and happen when a handful of objects or associations become extremely popular as they appear in the News Feeds of tens of millions of users at the same time.



There are two tiers of caching clusters in each geographical region. Clients talk to the first tier, called followers. If a cache miss occurs on the follower, the follower attempts to fill its cache from a second tier, called a leader. Leaders talk directly to a MySQL cluster in that region. All TAO writes go through followers to leaders. Caches are updated as the reply to a successful write propagates back down the chain of clusters. Leaders are responsible for maintaining cache consistency within a region. They also act as secondary caches, with an option to cache objects and associations in Flash. Last but not least, they provide an additional safety net to protect the persistent store during planned or unplanned outages.

Consistency

We chose eventual consistency as the default consistency model for TAO. Our choice was driven by both performance considerations and the inescapable consequences of CAP theorem for practical distributed systems, where machine failures and network partitioning (even within the data center) are a virtual certainty. For many of our products, TAO losing consistency is a lesser evil than losing availability. TAO tries hard to guarantee with high probability that users always see their own updates. For the few use cases requiring strong consistency, TAO clients may override the default policy at the expense of higher processing cost and potential loss of availability.

We run TAO as single-master per shard and rely on MySQL replication to propagate updates from the region where the shard is mastered to all other regions (slave regions). A slave cannot update the shard in its regional

persistent store. It forwards all writes to the shard's master region. The write-through design of cache simplifies maintaining read-after-write consistency for writes that are made in a slave region for the affected shard. If necessary, the mastership can be switched to another region at any time. This is an automated procedure that is commonly used for restoring availability when a hardware failure brings down a MySQL instance.

A massive amount of effort has gone into making TAO the easy to use and powerful distributed data store that it is today. TAO has become one of the most important data stores at Facebook -- the power of graph helps us tame the demanding and dynamic social workload. For more details on the design, implementation, and performance of TAO I invite you to read our technical paper published in the Usenix ATC '13 proceedings.

Thanks to all the engineers who worked on building TAO: Zach Amsden, Nathan Bronson, George Cabrera, Prasad Chakka, Tom Conerly, Peter Dimov, Hui Ding, Mark Drayton, Jack Ferris, Anthony Giardullo, Sathya Gunasekar, Sachin Kulkarni, Nathan Lawrence, Bo Liu, Sarang Masti, Jim Meyering, Dmitri Petrov, Hal Prince, Lovro Puzar, Terry Shen, Tony Savor, David Goode, and Venkat Venkataramani.

Excerpted from *TAO: The power of the graph*
https://www.facebook.com/notes/facebook-engineering/tao-the-power-of-the-graph/10151525983993920

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