Question2Answer learning notes – especially methodology dealing with huge number of users

1. Use words as counters

While recalculating, questions, answers and comments have wordids. And the concrete operation is to count their words rather than itself, which means that a certain entity is replaced by some words. It is not hard to imagine that when questions accumulate, storing the entire sentence becomes low efficiency, and wordid turns to be of great use.

1. Select favorites

Add the entity $entitytype with $entityid to the favorites list of $userid. Handles switching streams across from per-user to per-entity based on how many other users have favorited the entity. If appropriate, it also adds recent events from that entity to the user's event stream.

Add in the favorite for this user, unshared events at first (will be switched later if appropriate). See whether this entity already has another favoriter who uses its shared event. If not, check whether it's time to switch it over to a shared stream.

1. Count hotness and points

If an event is able to produce points, then set up a corresponding table for storage. Returns an array containing all the calculation formulae for the userpoints table. Each element of this array is for one column - the key contains the column name, and the value is a further array of two elements. The element 'formula' contains the SQL fragment that calculates the columns value for one or more users, where the ~ symbol within the fragment is substituted for a constraint on which users we are interested in. The element 'multiple' specifies what to multiply each column by to create the final sum in the points column.

Update the userpoints table in the database for $userid and $columns, plus the summary points column. Set $columns to true for all, empty for none, an array for several, or a single value for one. “This dynamically builds some fairly crazy looking SQL, but it works, and saves repeat calculations.”

Hotness is relevant with views. Hence recalculate the hotness in the database for posts $firstpostid to $lastpostid (if specified) If $viewincrement is true, also increment the views counter for the post, and include that in the hotness calculation

1. Set up categories for storage

Categories are created to show the path of querying. Also, if a question is altered, its category path will also change. This method is used to store lots of questions, and make them easily found.

1. Manage and truncate events

Question2Answer has two streams to store events, and one is for specific users, the other is for sharing. Then add an event to the event streams for entity $entitytype with $entityid. The event of type $updatetype relates to $lastpostid whose antecedent question is $questionid, and was caused by $lastuserid. Pass a unix $timestamp for the event time or leave as null to use now. This will add the event both to the entity's shared stream, and the individual user streams for any users following the entity not via its shared stream.

Because the shared event stream is related to effective query and update, there might not be too many tuples. There are two ways. One is to limit entries of a certain question entity. The other is to truncate the shared stream directly.

If the event is a kind of behavior rather than an entity, the event of type $updatetype relates to $lastpostid whose antecedent question is $questionid, and was caused by $lastuserid. Pass a unix $timestamp for the event time or leave as null to use now. Also handles truncation to avoid too many events for a question.

Trim the number of events in the event stream for $userid. If an event was just added for a particular question, pass the question's id in $questionid (to help focus the truncation). If that didn't happen, try truncating the stream in general based on its total length.

Why two streams?

An event stream is defined as the set of events which are thrown off ("published") by a particular entity. For example, it could include the activity on a particular question, or the activity by a particular user. We have an arbitrary many-to-many mapping between event streams and users subscribed to those streams. Over time, a particularly popular event stream could accumulate thousands of subscribers. Similarly, over time, a particularly hyperactive user could end up subscribing to thousands of event streams.

If we stored a single copy of each event stream in the database, publishing an event would be very fast. However retrieving a hyperactive user's update page would be extremely slow, because it would require retrieving all the streams they are subscribed to, and finding the globally most recent (e.g.) 50 events across all those streams.

So instead we could store a list of news updates for each user. In this case, retrieving a user's update page would be very fast. However, recording an event for a popular stream could become extremely slow, since it would have to be copied for every user subscribed to the stream.

The standard solution to these "publish and subscribe" situations is a message-passing architecture. That's what Twitter et al use. However that's not a viable option here, because it requires a process to be running in the background to manage the queuing and transport of these messages from publishers (event streams) to subscribers (users' lists of news updates). While we could have a cron-style process to manage this, I'm avoiding it for as long

as possible since it complicates setup. It also means there can be delays in updating users' news feeds.

So instead we adopt a hybrid approach. For each event created in an entity's stream, we record a single copy of that event in the entity's stream in the qa\_sharedevents table. In addition, by default, we place a copy of that event into the list of news updates for each user subscribed to the stream, via the qa\_userevents table.

However, if there are more than a certain number of subscribers to the stream, we skip this second step, i.e. we only record one copy in the qa\_sharedevents table. This limits the cost of publishing an event.

When we generate a user's list of recent updates, we of course retrieve the list of news updates for that user from qa\_userevents. However we also check to see whether that user is subscribed to any event streams for which updates are no longer posted into the user's own list, because the stream has too many subscribers. For each of these popular streams, we also retrieve the stream's events from qa\_sharedevents. Since users are only likely to be subscribed to a small number of popular streams, this limits the cost of retrieving the news updates.

(Having a shared event stream helps us another way. When a user subscribes to a stream, they can immediately have recent events from that stream copied into their list of news updates.)

Note that this approach isn't aimed at reducing the total cost of keeping all users up-to-date on all events, but rather ensuring that no individual operation (posting an event or retrieving a user's list of updates) takes too long, since that would turn into a very slow response time for the corresponding HTTP request.

What should we use for the threshold T, so that if a stream has more than T subscribers, its events are only recorded in the shared stream? One approach is as follows:

[this ignores stream length and truncation, which are constant factors]

T = our threshold

M = the maximum number of streams subscribed to by any user

P(x) = the probability that a particular stream has more than x subscribers

C1 = maximum cost of adding an event = maximum number of streams to which event must be added = O(T)

C2 = maximum cost of retrieving news updates = maximum number of shared streams to be combined = O(M \* P(T))

[we assume that the chance a particular user is subscribed to a particular stream is independent of the user]

Now if we assume the power law, aka 80/20 rule, we can estimate that P(T) is proportional to 1/T, so that:

C2 = O(M / T)

To minimize the maximum of these two complexity maxima, we want to equate them, so that:

T = M/T => T=sqrt(M)

So we could keep track of the maximum number of event streams any user is subscribed to, and use its square root.

Instead of that, we adopt an on-the-fly approach. We start by setting T=10 (see 'max\_copy\_user\_updates' in qa-app-options.php) since it's no big deal to write 10 rows to a table. Recall that whenever an event stream gets more than T subscribers, we switch those subscribers over to the shared stream. At that point, we check the maximum number of (total) shared streams that any of those users are subscribed to. If this is above T, that means that our maximum cost of retrieving a list of news updates is starting to go past our maximum cost of recording an event. So we rebalance things out by increasing T as appropriate, for use in future cases.