Proposal and work in progress

Oscar Moll-Thomae

October 28, 2011

1 Overview

There are three main types of operations expected of the graph store, or a graph store in general. One is querying for edge data `getEdge(v,w)`, another one is for fan-out `getFanout(v)`, another for intersections of the neighbrohoods of two nodes `getIntersection(v,w)`. There is also a tentative random walk one that I am yet to actually look into. A real workload is made of a mix of these queries, but I am thinking of partitioning strategies by starting with one of the operations and thinking backward to something that would improve their performance. For that reason, the benchmarks I have in mind are: 100% `getEdge()`, 100% `getFanout()`, 100% `getIntersection()`and a mix with realistic twitter proportions. .

TODO: put workload numbers from wiki here

The thoughts on strategies are here:

1. `getEdge()`: no strategy targets to optimize this in particular, but we should know clearly whether they perform well on this.

2. `getFanout()`: fanouts in a graph with wide variation in degree would improve with treating popular users differntly. The amount of benefit from this depends on several factors. A first factor is whether nodes with large degree have fanouts more often than nodes with smaller followings. Another factor is whether the rest of the system takes advantage of the parallelism in this optmized `getFanout()`.

   NOTE: I am not worrying about the rest of the system for my thesis, but in reality it would matter if the client does not process stuff in parallel. Also, looking at actual data from queries would help me 1) model simulated workloads, 2) talk about it in this text

3. `getIntersection()`: pairs that historically intersect often should be stored together.

   NOTE: again, i need historical data to effectively model this

4. `randomWalk()`: cluster actual graph, replicate.

TODO: talk to Raghavendra, Pankaj about this, and try reading more about it. This operation seems less important but kind of exciting, and would give room to looking at other kinds of work. Separate note: A stab at interface: could be some thing like `List<Pair<Node,Int>> randomWalkFrom(v, depth)` where hopefully depth is 2 or 3, for example. Also, this kind of query can return an approximate, potentially inaccurate result if this results in faster, online versions of it

This thesis presents three sections. The first is the design of three strategies based on the actual api operations we would like to support.

NOTE: Some iteration on design necessarily involves looking at real data (to get a better feel for what won't be effective). For example, I have used what Raghavendra emailed in September to evaluate viability for the `getFanout()`strategy.

Second is the definiton and results of simulated workloads on a bare bones version of a distributed graph db.This bare bones version is not intended to deal with failures nor with online resharding/rebalancing. In simulated workloads I have the option of changing things like skew, correlations and simulatied graph structure, the more I know about the real ones the better I can decide what to vary and to design the benchmark generator with it in mind. Finally, while the simulation gives the thesis a more general conceptual coverage because I can explore different parameters, sizes and hypothetical scenarios, the third section is the integration and benchmarking of the strategies on Twitter's graph service.

NOTE: right now I want to focus on getting infrastructure/benchmarks done for the `getFanout()`and `getIntersection()`before looking at the other stuff. This week I am trying to get to run a simulated workload good enough to make meaningful graphs about `getFanout()`from it and write them up in this document. You can check what I have done so far on that front in my repo, or on the review

NOTE: one nice distinction made in the Zynga techtalk was the difference between a social graph (ie follows that happened at any time in the past), and the 'active' social graph, where the edges disappear unless there is recent activity, this kind of query is of a lot of interest for monitoring growth/etc. How would a graph store facilitate this, if at all?, and, how does this tie in with looking at query logs to identify 'active pairs' and 'active users' (to optimize intersections and fanouts respectively. Also, recommendations that use the active social graph may be better than plain static social graph.

TODO: think about performance effects of doing these reads while there are updates happening

NOTE: Don't read the rest, its taken from the wiki, and its more of a text dump

2 Underview, don't read

1. design: I've fleshed out one alternative sharding function, and have a second one brewing. I want another method besides those.
2. implementation: on top of the graph service architecture (which is itself not implemented yet). For the past week and a half I worked on a simulator for the rest of the system that I will use to test and measure what I am doing. Eventually I'll get to try these things in the same data that the team will be testing their own new social graph store.
3. results: I expect to have a set of results be simulation based (eg simulated graph, simulated overall system), and one set based on actual graph data + actual overall system.

2.1 Challenges and goals

A challenge on the design side is a lot of the thinking done may not result in a significantly better scheme, and just provide significantly more complicated one with modest or no performance gains.

On the implementation side, the Data Services team is currently building this infrastructure so I also am working on a moving target, and it makes it more challenging for me to anchor implementation designs.

For an MIT MEng thesis it is probably enough to present a well informed and technically challenging exploration involving design, implementation and measurements. For Twitter I'd like to contribute something that at least informs the design, but at best provides a viable sharding mechanism specialized enough for the twitter graph that it performs better than general methods and can be used in production.

3 Design

3.1 Background

Note: some of the numbers (eg on cost of reading all followers, correlation between frequency of fanouts and edge set size for a node) will need updating as I learn them.

Sharding in the graph data store is proposed to be hashing by left endpoint. ie a mapping like (A,B) -¿ A -¿ shard_id. This way of sharding guarantees edge localy because all edges with the same left vertex shard to the same sharding unit. There is another design proposal that is also hashing based, but it brings bits from both h(A) and h(B) together to make the mapping less random than a hash by full (A,B). The effect of this scheme is to limit the number of shards all edges of the form (A,*) map to.

In any case, there are two immediate and competing interests. One is minimizing load imbalance, the other minimizing the costs of distribution. (ie one-shard is one extreme, all-shards queries are another one). The 'optimal' scheme from this load point of view depends on actual skews of both the actual degree distribution for the graph as well as the skew in queries to particular pieces of data, and also the relative frequencies of single edge queries (does A follow B) vs multiple edge queries (who follows A). For example, if single edge queries get(A,B) are overwhelmingly more common than get(A,*) then locality is not as important, and higher numbers of messages across multiple nodes could be tolerated. As another example, if a lot of individual single edge queries go to edges from the same left vertex A, then spreading them out across shards is best. Similarly, if nodes have many many followers, then serving everything from a single shard may hinder parallelism. All of these questions depend on the empirical structure of the graph and queries.

The proposed sharding by left node is a simple, clean option. There are throughput targets for different operations, as well as target latencies that need to face these uneven distributions. So a good question is whether sharding by left vertex is good enough. At the 99.99 percentile, the number of followers is 25000, which means that if a single shard is to hold all of them then a query of the form 'get all followers for A' requires (2.5 e 4 edges) * (1 e 2 Bytes/edge) = 2.5 e 6 B = 2.5 MB (x2 to allowing for some memory overhead, besides replication). Reading this sequentially from Memory takes 250 usec/MB * 5 MB 1 ms. Which is well within the 10 ms bounds. On the other hand there are nodes in the distribution tail that right now have 10M edges. The same read delay calculation becomes 1 e 7 edges * 1 e 2 bytes/edge = 1 e 9 B = 1 GB, and allowing for some memory overhead, 2GB. 500 ms, by far past the limits. An inverse calculation can be used to find the 'cutoff' at which simply reading from memory is already a bottleneck. This comes to 200k edges, (percentile 5 nines).

The argument about reading from memory suggests it is not possible to meet these targets, but there are several counter-arguments that dampen its implications. The first one is that these are the long tails of the degree distribution, so we can allow them to take longer and still meet targets. The second one is that in reality the 'get all followers of A' type queries are a smaller fraction than initially apparent: the applications are already designed to make these requests in smaller batches. The third one is that the calculation above points out that memory reads won't be done as fast, but a lot of other things will be worse than that much earlier. The network bandwitdth is presumably lower, and even capacity of clients to consume the data. Moreover, this kind of query for 'all followers of A' will become less common if a proposed 'selective timeline materialization' strategy becomes used by the rest of the system. There are counterarguments to these points too. One is that this small but influential group of users is of some value beyond simply optimizing for 99.99% of the cases would reflect. But even strictly based on the numbers. If we mean 99.99 percent of individual ¡requests¿ must be below 10ms, then the fact that queries to these particular 0.001% of ¡users¿ are may actually translate to violating the SLA depending on how many queries go to this particular group. (Data on the correlation between query rate and degree is still forthcoming).

Going deeper in the correlation reasoning of the previous paragraph. There are potential interdepencies in 'skew' that I have not considered but could be maybe worth taking a look at in data. Hypothetically, there could be dependencies in the the degree frequency skew and the query frequency for a particular or node edge. For example, the users with the most followers are also likely to be the ones growing the fastest, so insert queries to their nodes may be more frequent. It may also be the case that users with more followers tend to tweet more often, so fanout queries asking for all their followers are executed more often. While these users may be a smaller fraction, the overall percentage of requests made for them is probably larger than 0.001%.

There are a few more observations of these types that could be investigated on the actual use logs, and that could be relevant to any design, and a potential part of the project. It could also be these things are just not significant, I'm not sure. While the complexity of this reasoning, the system designs resulting from it need not be complex.

3.2 Proposal

Here are a couple of approaches I have in mind:

3.2.1 Two-tier sharding:

This is my simpler, and better thought out method right now. It is influenced a lot by the fine grained partitioning paper in that it exploits the possibility of using explicit lookups in some cases, and a default for others.

The idea in this method is to classify vertices (UIDs) in 2 tiers based on the number of edges and then treat them differently based on tier. For example queries for users with 0 - 100k followers would be best served by storing all edges together in the same node, but a method such as hashing (including clustered consistent hashing) would distribute them among several. Users with ¿ 100k (100k is the cutoff order of magnitude where it is still possible to read fast in a single node) In the current graph there are less than 3k users with this number of followers, so we can easily store an explicit lookup table (uid -¿ replica set) for that kind of user, or simply use a different clustered consistent hashing scheme on this set only since here it is expected we distribute work among nodes. The explicit lookup table still can be done even if the graph grows to 500 M while keeping its degree distribution about the same. This kind of sharding should improve performance for queries like get(A,*) without really hurting single edge queries or anything other query in particular (at the price of the overhead of moving things from one tier to another sometimes).

There are several issues to solve in this case. One is to still be able to answer queries by edge. For a vertex A in the 'power' user tier, if we query for (A, B) we won't immediately know where it is.

Also, different shard units for this same user need to go in different machines, so that we can read their edges in parallel, but this won't be really helpful if the rest of the system serializes going through this sequence. This could be simulated, though.

Another issue is dealing with the changes in the user graph. There are scenarios where a user moves from one tier to the other. This involves something akin to range splitting (but potentially on a user by user basis) as happens in range partitioning. When a node goes down this can probably be dealt with just as with the other schemes. Two tier sharding can be expanding to more than 2 tiers, and how we pick who goes in which tier can be done based not only on degree but arbitrary policies.

3.2.2 Social sharding:

This method is more based on the one explained in the Schism paper. Here I will first look at which pairs of users are often used in intersection queries, then I will construct a graph based on this just like in schism, and try to shard these users together. This way, intersection queries can be pushed down to the nodes.

I have to think about whether other ways to partition this (that use social graph information of some sort), are also good idea.

The graph exploration and partitioning can be done offline, but we still need to have a concise representation of the look up table available to do the online lookups express the results of any offline algorithms, this is a challenge I have not really come up with a solution for.

Additionally there is a problem if the graph structure changes a lot over time, ie how do we keep the sharding up to date if every time we update it it may change enough that a lot of work is needed to update it. Still, the fine grained partitioning paper offers some options for concisely representing /compressing the look up table in some cases. The advantage here is that for queries such as intersect_edges(A,B), if this kind of query occurs often when A follows B, then by splitting the graph this way we gain the advantages of not just hashing by left-endpoint, but looking up edges(A, *), and edges(B, *) and the subsequent intersection can be done at the node, and only the result set sent back to the API server.

Lastly, besides the the two main queries, the are other types of queries such as "get(A,) Union get(B, *)" or "get(A,, follows) Intersection get(B, *, follower)". This last query occurs for example when a user visits the timeline of another. Since presumably this happens more for users that follow one another, then encouraging the placement of adjacent nodes in the same shard enables us to push intersections into single nodes a larger fraction of the time. There are also more experimental kinds of operations that do not exist at this point such as explore 2 hops away.

3.2.3 other

There are other important families of queries I have not though as much about, such as time based queries such as get_follers(A, within last 3 days). these offer other areas to think about. Other ideas are

4 Implementation

4.1 Targets

1. Implementing hashing strategies (currently implementing the two tier strategy)
2. Working around the current sharding library limitations as I realize them, and making sure I understand the interface to it. (In progress)
3. Implementing some simulation classes to benchmark my stuff independently of whether there is a fully working system (also just for testing). (In progress)
4. Implementing the other sharding strategies (ie some may require offline learning of some graph properties), may include using external tools to generate some graphs and partition them.

4.2 Benchmarks

4.2.1 Background

The goal here is to measure how much better or worse these perform, after fixing the number of machines, based on the empirical patterns found earlier in the design part. Workloads could be derived from real operation data or hypothetical data with skews like done in the PNUTS paper. Reading from different papers how they benchmark things would help.

4.2.2 Proposal

To benchmark the two-tier hash strategy

1. Will want to see how the viability of the strategy (eg amount of state needed for lookup table) changes as the graph changes. Eg. in simulation vary the degree skew.
2. Will want to vary the workload composition (Eg % of fanouts, intersections)
3. Will want to quantify time and space cost of running the custom shard code vs. will want to quantify performance of the full graph data system with custom shard.
4. Will want to quantify cost of having to move data around every now and then (since you need this when you don't hash) in the case of a strategy where I need to crawl the graph offline, benchmark the process itself.

Another way to compare methods is by comparing resources needed to achieve the same performance requirements. (vs comparing the results achieved with the same resources).

Finally, factors such as bottlenecks in the rest of the system may hinder observing the effects desired. I can modify the simulator to show that, but could affect other results.