Database Partitioning Strategies for Social Network Data

by

Oscar Ricardo Moll Thomae

B.S. EECS, Massachusetts Institute of Technology (2011)
B.S. Mathematics, Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Masters of Engingeering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2012

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2012

Certified by . . . .
Stu Hood
Engineer at Twitter
Company Supervisor

Certified by . . . .
Samuel R. Madden
Associate Professor
MIT Thesis Supervisor

Accepted by . . . .
Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Database Partitioning Strategies for Social Network Data

by

Oscar Ricardo Moll Thomae

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2012, in partial fulfillment of the
requirements for the degree of
Masters of Engingeering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed, prototyped and benchmarked two different data partitioning strategies for social network type workloads. The first strategy takes advantage of the heavy-tailed degree distributions of social networks to optimize the latency of vertex neighborhood queries. The second strategy takes advantage of the high temporal locality of workloads to improve latencies for vertex neighborhood intersection queries. Both techniques aim to shorten the tail of the latency distribution, while avoiding decreased write performance or reduced system throughput when compared to the default hash partitioning approach. The strategies presented were evaluated using synthetic workloads of my own design as well as real workloads provided by Twitter, and show promising improvements in latency at some cost in system complexity.

Thesis Supervisor: Stu Hood
Title: Engineer at Twitter

Thesis Supervisor: Samuel R. Madden
Title: Associate Professor

Acknowledgments

On the MIT side, I thank professor Sam Madden for accepting being my advisor in the first place, meeting online and in person and providing feedback on my work and guiding it with useful suggestions. Thanks also to Dr. Carlo Curino for giving me ideas and meeting on Skype to discuss them. On the Twitter side, thanks to the company for letting me work there during the Fall of 2011, to Judy Cheong for helping me join the Twitter MIT VI-A program last minute, to the Twitter Data Services team for being my hosts during the fall of 2011. Thanks especially to Stu Hood for taking up the supervisor role, meeting regularly, providing resources for my work and dicussing ideas.

Contents

List of Figures

List of Tables

Introduction

Twitter is a popular web service and social network that anyone can join. Once in it, users create a profile and follow other users. Users can post brief messages, known as Tweets, that get delivered in real time to their followers. Similarly, users can view the latest messages from the users they follow.

As the network has grown to hundreds of millions of users, the backend systems supporting Twitter have developed into a vast custom infrastructure, offering many exciting opportunities for computer systems research. In particular, the data stores are responsible for storing all the user data as well as serving it quickly upon request. These services need to reliably store billions of relations between users, their profiles, their messages, and are tasked with serving this information fast enough to support the almost instant delivery of up to 30000 unique Tweets per second [10] to up to about 20 million users in some cases. This kind of scaling requirement is a strong motivation to seek alternative methods of serving these specific workloads.

The data stores, or data services in Twitter parlance, are among the core backend systems supporting the site. One of these data services, the graph store, is in charge of storing the relations between users. The graph store supports basic graph data structure operations such as inserting and removing vertices and edges, and retrieving adjacency lists for vertices. In addition to that, it also supports more complicated set arithmetic queries such as intersections, counts and differences. This thesis proposes improvements to the graph store.

The techniques proposed are designed for Twitter's specific workload, rather than for general purpose database systems, but these techniques generalize to other social network type workloads. Social network type workloads are characterized for their extremes. For example, in Twitter some users are much more popular and central to the rest of the community than others. Operations on them benefit from being treated differently.

In chapter 1 I survey the general purpose partitioning approaches used in other systems and how they related to query. In chapter 2 I look at the specifics to Twitter's workload. Knowing the specifics of both the kind of data stored by Twitter's graph store and the patters in its query workload help motivate proposals to improve the specific Twitter case. In chapter 3 I introduce the two proposals and quantitative arguments to explain why they should work. Finally, in chapter 4 I present the results from prototyping and benchmarking these techniques on both synthetic and real data provided by Twitter.

Chapter 1

Background

Many systems with high throughput needs such as Google, Facebook or Twitter achieve their scalability goals by building distributed infrastructure. This is the case for most of their infrastructure, from the web servers and caches down to the databases backing these services. In the specific case of databases, the potential gains from using multiple computers simultaneously depend partly on a good partitioning of the underlying data. Database partitioning in general is a widely studied topic, and there are several standard solutions that work well in general but are not optimal in the specific case of social network workloads. This thesis presents and evaluates two different partitioning strategies for social network type workloads.

There are two basic properties desirable from shared nothing distributed system. The first property, *scale out*, is that we can double the throughput of the system simply by doubling the number of nodes, without affecting latency. The second property, *speed up*, is that we can halve the latency by doubling the number of nodes. In practice, it is common that doubling the number of nodes in a system will less than double its throughput, or less than halve the latency according to the objective, but those are still the ideals. For websites such as Twitter scale out is seen as the primary goal because it implies that an increase in users can be matched by a proportional investment in machines, and that the average cost of running the site remains the same. Latency is also important from the user experience perspective: the web page needs to feel responsive, and messages sent from user to user should arrive almost instantaneously.

Traditionally, workloads fall into two categories. The first category is online transaction processing, or OLTP, in which the read and write sets of an operation are fairly small. The second category is online analytical processing, or OLAP, in which the read sets are fairly large. The emphasis on the first category is on ensuring scale out, while for the second category there is also attention to speed up. Very often, the databases serving websites are considered OLTP, and optimized for throughput. But because in social networks such as Twitter some users have a lot more edges than others, then for such users, queries like counts and scans are in fact a lot more similar to OLAP. This type of extreme workload variation is one of the key differences from other typical workloads, and why speeding up some of the queries becomes important. In the next section we wee how different data partitioning policies influence a system's scaling properties.

1.1 Database partitioning

Ideally, a partitioning strategy enables scaling by distributing all work evenly across the nodes and having each node work independently of the others. When the work is not evenly distributed among nodes then the overloaded nodes will likely show reduced throughput, thus reducing overall system throughput. There are reasons why work may not be perfectly distributed, one is the unpredictability of which items will be popular. Similarly, communication and coordination across nodes mean there will always be some degree of dependence between nodes. When the nodes need to constantly communicate data or coordinate updates to ensure they are all or none, like in distributed transaction, then the extra waiting due to locks may reduce throughput. Even if there is no locking, all the communication between nodes may turn network bandwidth into the bottleneck. Some operations invariably require looking at several tuples at the same time, and these may be located in different physical computers.

The most general purpose partitioning strategies are hash and range partitioning [6]. Like their name suggests, hash partitioning uses a hash function to map rows to nodes, while range partitioning defines value ranges and then maps each to a node. Given an suitable hash function, hash based partitioning on any sufficiently diverse attribute will achieve the goal of balancing tuples and requests evenly, even as the table changes. On the other hand, hash partitioning only guarantees to map equal values to equal nodes. Two different values will be mapped to the same machine only by chance no matter how similar. With probability $1/n$ for a cluster of n machines, which tends to 0 as n grows. Range partitioning, on the other hand, will map rows with values close in their natural ordering to the same machine, achieving locality. Range partitioning may spread things less equally than hashing and will need to repartition in case one range turns out to be a lot more popular than other ranges, but its locality properties may help reduce the competing cost of communication across nodes when queries themselves tend to read nearby values of attributes. This thesis proposes two alternative partitioning methods that are not intended to general purpose, but that should work better for Twitter graph store workloads, explained in more detail in chapter 2.

We saw what methods we can use to partition, but another aspect of partitioning is which attributes we decided to partition by. Parallel joins are a typical example. If two large tables are expected to be joined regularly on a field x, partitioning both tables using the same policy on field x allows the join to proceed in parallel without need to reshuffle one of the tables across machines. In this case, there is both a throughput gain (no communication) and a latency gain (parallelism). Another example illustrates the interactions between choosing partitioning attributes and partitioning methods. If we partition a table of log entries by using timestamps and a range partitioner, then only one machine will be in charge of receiving all updates at one time and can be overloaded. On the other hand, if we choose to partition by a hash on the timestamp the write load would be spread across the whole cluster.Similarly, if we had chosen to partition by a range on a different attribute such as log even category, we would also more likely have spread the workload. The moral is the partitioning decisions need to be made based on knowledge of the workload.

The closely related concept of database replication deals with how many copies of each block of data to keep and where to keep them. Even though replication is often used primarily to ensure fault tolerance, replication decisions also can help improve performance. For example, a system may replicate heavily read data, allowing the read requests to be served by different machines. On the other hand this replication policy increases the cost of writing an update. Depending on the read to write ratio of the data, this policy can increase or decrease throughput. Another example of using replication for performance purposes is when a database replicates a table thats frequently joined with others, this policy helps avoid communication overhead, and again, involves changing the cost for updates. In this thesis, the aim was to not affect writing overhead, so we do not consider alternative replication policies, mostly focusing on the partitioning itself.

Different partitioning policies involve two different implementation challenges. The first challenge is in storing enough information to enable efficient lookups. Hash partitioning is very practical from this point of view, because a hash function contains all the information needed to map any tuple to its location, the overhead for both storage space and lookup time is $O(1)$ relative to the number of tuples in the database as well as the number of nodes, this is relatively little state, so we could even place it in the client library directly. Range partitioning is not as easy, it involves keeping a range directory structure that grows as $O(r)$ where r is the number of ranges we have split the tuple universe into, because of that, it is less practical to place the state in the client library and we may need a partition manager node in the architecture. Alternative partitioning systems, such as those proposed in this thesis, also need their own customized data structures for implementation. The storage challenge is complicated by system needs to scale up and tolerate node failures, which is the second challenge.

The more nodes participate in a database, the more likely it is some will fail, and also the more we need to support easy addition and removal of nodes. The partitioning infrastructure must make it possible to add new nodes without creating too much work to adapt. For example, if our partitioning function were $f(k) = \text{hash}(k) \bmod n$ where n is the number of nodes, then the load would be well balanced on a static system. On the other hand, if n changes then the system must move most of the data from from their old location to the new one. There are more elaborate techniques to deal with these problems, such as extensible hashing and consistent hashing, but they are more complicated than in the static case. Consistent hashing, for example, requires an ordered list of size $O(n)$ where n is the number of nodes. It also requires to be updated whenever there is an event such as a node leaving or entering the system. In addition to dealing with nodes that enter or leave the system, range partitioning must react to many tuples entering or leaving the same range, and react by splitting or merging ranges and relocating data accordingly. As routing information grows, the system needs to be able to recover it after failures and to keep it consistent across different servers and between servers and actual nodes storing the data. All of these are important problems, but not the focus of this thesis.

1.2 Graph oriented systems

In section 1.1 I described partitioning from the general point of view of databases. This thesis is specifically concerned with partitioning policies for the Twitter graph store. The graph store does fall within the domain of databases, but has a more specific interface and workload. In the case of the graph store, the data is users and their relations, and the operations are adding users and relationships between users, and reading them too. I explain the interface and workload in more detail in chapter 2.

Graph workloads are a specific niche of database workloads, but they are not unique to Twitter nor to social networking applications. Geographic entities such as roads and cities, semantic web relations and graphical models in machine learning all exhibit graph structure and can be processed as such. Similarly, Twitter's graph store is not the only one graph data store. There are many other systems specifically designed to store and process graph data. Ultimately, a graph can be stored in a typical relational database and queried there, too. For example, we can store a tables of vertices and tables of edges. Operations such as neighbors, queries such as degree per nodes, 2-hop neighbors can be expressed etc can all be expressed with typical SQL statements. Nevertheless, graph systems with custom interfaces and organization are becoming popular [24] [31].

The graph processing systems described can be split broadly into two categories: analytics and storage. Graphlab [21], Pregel [22], and Cassovary [16] work for the analytics category. Graphlab and Pregel are distributed, and Cassovary is not. All of them offer a graph-centric programming model and aim to enable efficient execution of richer algorithms. On the other end there are the storage systems, these include the Twitter graph store (also known as FlockDB)[17], which provides distributed and persistent storage with online reads and writes and offers a much more limited interface: get sets of neighbors, get edge information for a specific edge and a few basic set operations. Other stores such as Neo4j sit somewhere in the middle: Neo4j offers persistence, online updates and a richer interface for queries: shortest paths, 2-hop neighbors, etc, but is not distributed [24].

Finally, partitioning in graph oriented systems is done similarly to that of databases. Pregel uses a function of the hash(ID) mod N as the default partitioner, for example, but allows to plug in different partitioners [22]. Apache Giraph is similar [3].

1.3 The abstract partitioning problem

I have introduced the partitioning problem in databases, why it is relevant to graph systems and how it is normally solved. Before presenting any proposals for better partitioning in the Twitter graph store, it is worth noting that the partitioning problem is complex algorithmic problem for which no solution is known. Partitioning problems, even in offline batches are often hard to solve optimally. Here I present two NP complete problems that show, in one way or another, that balancing problems can only be dealt with heuristically. The first problem is about partitioning a set of numbers of different values, the second problem is about partitioning a graph with arbitrary edges.

Definition 1. *Balanced partition problem.* input *A group of n positive numbers $S = a_1, ...a_n$*

output *A partition of the numbers into two groups A and S A such that $|\sum_{a_i \in A} a_i - \sum_{a_j \notin A} a_j|$ is minimized.*

This problem can be interpreted as a way to optimally balance the load across servers if we knew ahead of time the load a_i for each task. In that case, a scheduler could run the algorithm and then allocate tasks accordingly. One interesting aspect of this problem is that it involves no communication among the parts, but already is NP-complete. For practical purposes, there exist several quick heuristic methods to generate a (non necessarily optimal) solution to this problem.

Often, data partitioning problems also need to consider communication costs. If doing task a_1 and a_2 involves some sort of communication, then we may want to group them in the same partition. Problems like such as this one, with an added communication cost can be better modeled as the graph partitioning problems rather than set partition problems. Formally speaking, the graph partitioning problem is the following.

Definition 2. *Graph partition problem* input *A graph $G = (V, E)$, and targets k and m.*

output *A partitioning of V with none of the the parts is larger than $|V|/m$ and with no more k edges crossing between parts.*

The graph partitioning problem, unlike the balanced partitioning problem, does not necessarily have weights, nevertheless it is NP-complete. A more general version allows both vertex and edge weights is therefore just as hard. Like for the number partition problem, there are efficient (but, again, not necessarily optimal) heuristics to solve it. One such heuristic is the Kernighan-Lin method that try finding a good solution in $O(n^2 \lg n)$ time [19]. Kernighan Lin is also one of the components of the more general METIS heuristic [25]. METIS is an algorithm and associated implementations that aims to solve larger scale instances of these problems. Graph partitioning algorithms such as these have been used for parallel computing applications as a pre-processing step, to divide the load evenly among processors.

The problems and heuristics presented in this section were results on the static (see everything at once, and graph does not change) case. One challenge for these algorithms is to scale to larger graphs. As the input graph size increase it may no longer be possible to fit the graph in a single machine. Another challenge for graphs that are constantly changing is to compute an incremental repartitioning that does not move too much data across parts and yet is effective. Running a static partitioner periodically may not be a good solution if the algorithm produces widely different partitions every time. Despite the challenges, there exist heuristics for distributed streaming graph partitioning [28].

Even though the partitioning problem is hard, a partitioning strategy does not need to achieve optimality. In the previous subsections we saw that hash partitioning is widely used despite causing many edges to cross between parts, so many other heuristics will probably perform better. For this thesis, one of the strategies proposed makes use of the METIS algorithm.

In this chapter I explained the relation between partitioning and scaling, listed the typical partitioning strategies supported by databases and their more recent relatives known as graph processing systems, and described known hardness results that are relevant to the partitioning problem. The next chapter will expand on the Twitter specific aspects of the project.

Chapter 2

Twitter specific background

The main persistent Twitter objects are users, their Tweets, and their relations (known as Follows) to other users. Each of these objects has associated information such as user profile picture and location in the case of an individual user, Tweet timestamp and location for a Tweet, and the creation time of a Follow between two users. Any user can choose follow any other user freely, there is no hard limit. Conversely, a user can be followed by anyone who chooses to. That policy combined with the growth in Twitter's popularity made some users such as the singer @ladygaga or current US president @BarackObama very popular. Each of them has about 20 million followers as of mid 2012. A note on terminology: Twitter usernames are commonly with the '@' symbol. Also, by *Followers* of @A we mean all users @B such that @B Follows @A. Conversely, the term *Follows* of @A refers to all the users @C such that @A Follows @C. A user chooses her *Follows*, but not her *Followers*. The relation is asymmetric.

Each Twitter user gets a view of the latest Tweets from his Follows, this view is called his Timeline. If @A Follows @B and @B Tweets c, then @A should see c in its timeline. Constructing all users' Timelines is equivalent joining the Follows table with Tweets table. At Twitter, the three tables: User, Tweet and Follows are implemented as three different services. So we talk about each table as the User store, the Tweet store, and the Graph store respectively. These databases are constantly joining entries to construct timelines, so all the operations involved in these process are crucial to Twitter's functioning. This separation also impacts the amount of user information available internally to the graph store, for example, it is not possible to simply partition users by country or by language because this information is not really part of the graph store, which deals with users only as user ids.

A second note on terminology: since the the graph vertices correspond to users, we treat the term vertex; and user as synonyms. Similarly, a Follow relation and an edge are also synonyms. The term 'node' is reserved for the physical machines on which the system runs.

2.1 Graph store interface

For the purposes of this thesis, we can think of the graph store as a system supporting three kinds of updates and three kinds of queries. The updates are to create an edge or vertex, to delete an edge or vertex and to update an existing edge. The three queries are described below. In short, the graph store can be thought of as offering basic adjacency list functionality, with additional support for some edge filtering and intersection.

The first basic query, known as an edge query, is a lookup of edge metadata given the two edge endpoints. Metadata includes timestamps for when the edge was created, when it was last updated and other edge properties such as the type of the edge. For example, it could be a standard 'Follow', or a 'Follow via SMS' that specifies SMS delivery of Tweets, or a 'blocked' edge. Edges are directed. At the application level, the edge query enables the site to inform a visitor of its relation to the profile he's looking. In the case of blocked users, the metadata also enables the system to hold their Tweets from being delivered to the blocker.

The second query, mentioned in the introduction as the list of followers query, is referred to as a fanout query. A fanout query returns all the neighbors of a given vertex @A. The complete interface to the graph store allows for some filters to be applied before returning results. For example, we may wish to read all the Followers of @A, or all the Follows of A or users blocked by @A. Some of these queries may involve filtering out of elements. The fanout query interface also offers to page through the results of a query, useful in case the client is not prepared to deal with large result sets, or simply because may be interested only in 1 or 10 followers. Such queries are used for display in the profile web page, for example. The paging interface requires specifying both a page size for the result set and an offset from which to resume. The ordering of the result set can be set to be the order in which follows happened. This provides a way to answer queries like 'who are the 10 latest followers?'. Because the time ordering of the edges probably differs from the natural order of the user ids, some of these queries involve more work than others. The most important use case of fanout operation is Tweet routing and delivery. Whenever user @A Tweets, the effects of that action must be propagated to the followers returned, which we find out using a fanout query.

The third query is the intersection query, the intersection of neighbors for two given vertices. Intersection queries are more heavy weight than fanout queries. A single intersection operation implies at least two fanout type operations plus the added work to intersect the results. The extra work can be substantial, for example if the two fanouts are coming from separate machines and the are not sorted by user id to start with. One design option is to let the server implement only fanout queries, and have the clients intersect the results themselves. This approach reduces work at the server, but increases external bandwidth use to transmit results that eventually get filtered anyway. Like with fanout queries, there are many parameter variations on this query. Like with fanouts, the result for an intersection can be paginated.

There are several Twitter use cases for the intersection query. When a user @A visits @B, the website shows @A a short list of his Follows that in turn are Followers of @B. This is an intersection query of the Follows of @A and the Followers of @B.

Because the website only displays the first 5 or 10 names, this use case only requires a executing part of the full intersection. The most important Twitter use case of the full intersection is, like with fanouts, also related to Tweet routing. When a public conversation takes place between two users @A and @B, Twitter propagates the conversation only to Followers of both @A and @B, so it must intersect Followers of @A with Followers of @B. There are potentially many uses for intersection query, such as computing a similarity metric between users such as cosine similarity, or measure the strength of the connection between two users @A and @B by intersecting @A's Follows with @B's Followers and counting the result. Another example is in listing paths: a full intersection of @A's Follows with @B's followers encodes all the 2-hop paths from @A to @B. An intersection query could also be used for looking more complicated path patterns in the graph, like follow triangles. or simple statistics: for example, how many mutual Follows-Followers does @A have? We can answer this by intersecting Follows @A with Followers @A. Intersection is the more complex operation considered in this thesis, and given these use cases it is possible that gains from improving its performance may enable the graph store to handle more interesting queries.

The graph store interface supports more than fanout and intersection queries. It actually supports more complicated set arithmetic such as three way intersections, and set differences. It also allows counting queries. A general optimization strategy for general set arithmetic queries is out of the scope of this thesis, and we will see that empirically the main uses of the store are fanouts and intersections.

There is limited need to enable arbitrarily powerful queries in the graph store because Twitter has also developed a separate graph processing system called Cassovary (mentioned in chapter 1). At Twitter, Cassovary is used for more analytic workloads. Unlike the graph store, Cassovary loads data once and then mostly reads. Because of the read only workload, data can be compressed much more. Because it is not meant to be persistent, all of it can be stored in memory. Even at the scale of Twitter's operations, the graph in compact form can fit into a single machine. As an estimate, the connections between 1/2 billion users with a combined degree of 50 can be represented efficiently in around 100GB. The kinds of queries done in Cassovary include things like page rank and graph traversal, and it is used to power applications such as Follow recommendations, similarity comparisons between users and search. These operations can use data that is a bit stale, since they are not meant to be exact. By contrast, the graph store cannot fall behind updates. When a user follows another, this change should be reflected in the website immediately. New Tweets from the recently followed user should be delivered as soon as there is a subscription. Similarly, when a user blocks another the system should react immediately. This is not the case with recommendations, which can be computed offline and more slowly.

Both FlockDB and Cassovary are open source projects, so full details of their interfaces and current implementations are available in the source repositories [17] [16]. While the interface description tells us which operations are possible, viewing actual logs informs us of which is the actual use the system. The next sections profiles the queries and data stored in the system..

2.2 Workload profile

The graph store workload is made up of a mix of the queries for edges, fanouts and intersections as well as by the data stored in it. By checking operation frequencies we gain a clearer picture of how the API is really used: which operations are relevant and which are not. And looking at the graph itself we understand the variations in work needed by these operations. This section clarifies the workload and helps understand what a social network workload is like.

The goals of analyzing the workload are to check that the operations merit optimization, to learn about the average and extreme cases, and to possibly learn facts that may help us devise improvements.

1. What are the relative frequencies of the queries?. How significant are fanouts

and intersections to the load on the graph store?

2. Are queries are made evenly across users or not? Are there any users with many fanout queries? Are there users that are intersected often?

3. Are users that get queried for fanout often also queried for intersection often?

4. What is the distribution of data among users? We know some users have reached 20 million followers while others have only about 50, are there many such cases? can these large follower counts be considered outliers or are there too many of them?

5. Is there any relation between a user's query popularity its degree?

To answer these questions I used a sample of all incoming requests to the running graph store for a few hours, resulting in about 300 million samples as well as a separately collected snapshot of the Follows graph. These logs did not sample operations such as writes, deletes and updates, nor counting operations, so we are limited to only comparing among queries.

2.2.1 Query profile

The short answer to the question about the frequencies of the different operations is that fanouts are the most popular queries, comprising about 80% of all queries. As described in section 2.1, the graph store supports paginated results for fanout queries, and so it accepts both a page size and an offset as part of the fanout query arguments. By analyzing the more common page sizes and offsets, I found that the main form of fanout query is only querying for a small page size starting at offset zero. The second most popular fanout query also starts at offset zero, but requests page sizes of above 100. As described in section 2.1 small page size fanout queries may be produced by user page views. Fanouts done with larger page sizes are more consistent with the Tweet routing use case. The full aggregate results are shown in Table 2.1.

operation type	frequency
fanout (small page, zero offset)	70%
fanout (large page, zero offset)	10%
intersection	1.5%
edge metadata	0.5%
other fanouts, etc	18%

Table 2.1: Graph store usage statistics from query logs

The ‘other fanouts, etc’ category includes fanouts done at starting at larger offsets and a few unrelated operations. Because a full logical fanout query may be actually implemented as a series of paged requests at different offsets, the ‘other fanouts, etc’ category may include the same logical fanout repeatedly, which is why I show it separately. And about the other queries, intersections happen much less often, suggesting they may not be as important. Surprisingly, edge metadata queries seem to happen seldom.

The significance of a particular type of query in the workload is not just a function of how frequent it is, but is really a product of its frequency and the load each individual operation places on the system. For instance, the page size given to a fanout query has a relation with the potential cost of the operation. Requesting the latest 1 or 10 followers is lightweight, but requesting 1000 of them can involve more work both in scanning it (it may occupy multiple pages on disk) and in sending it (its more data after all). For this reason, I believe they should be considered separately. Similarly, a single intersection operation implies at least 2 fanouts and at some merging (which can be quadratic in size). Even at small page sizes, the intersection operation may need to internally read much larger fractions of the fanouts in order to compute a small number of answer elements. so, the effect of intersection on system load could well be larger than its frequency alone suggests.

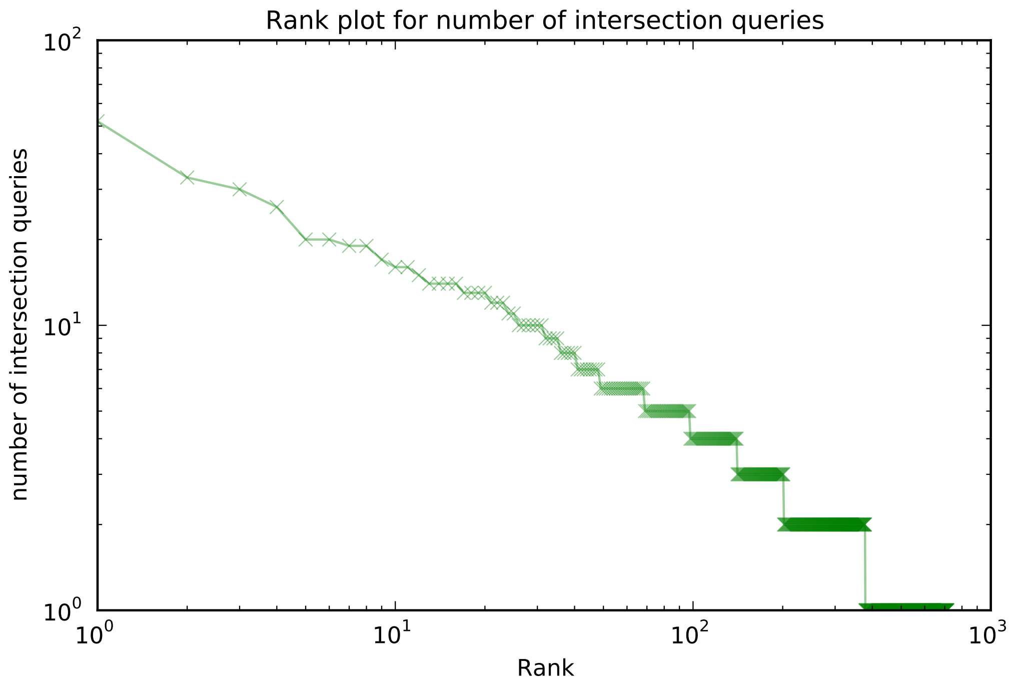
The effect of this load factor can easily match the effect of frequency: since the light weight fanout operations represent at most about 70% of the requests, and the heavy weight ones about 10%, then an individual heavyweight operation needs to be about 7$\times$ more work than a single light weight, for the heavyweight operations become just as significant as the light weight ones. I did not have a way of checking whether this difference in workload is as significant as this or not. Either way, as a result of this observation the work in this thesis focuses on the heavy fanout queries and slightly less on the intersection queries, and none on the edge queries.

2.2.2 Query skew

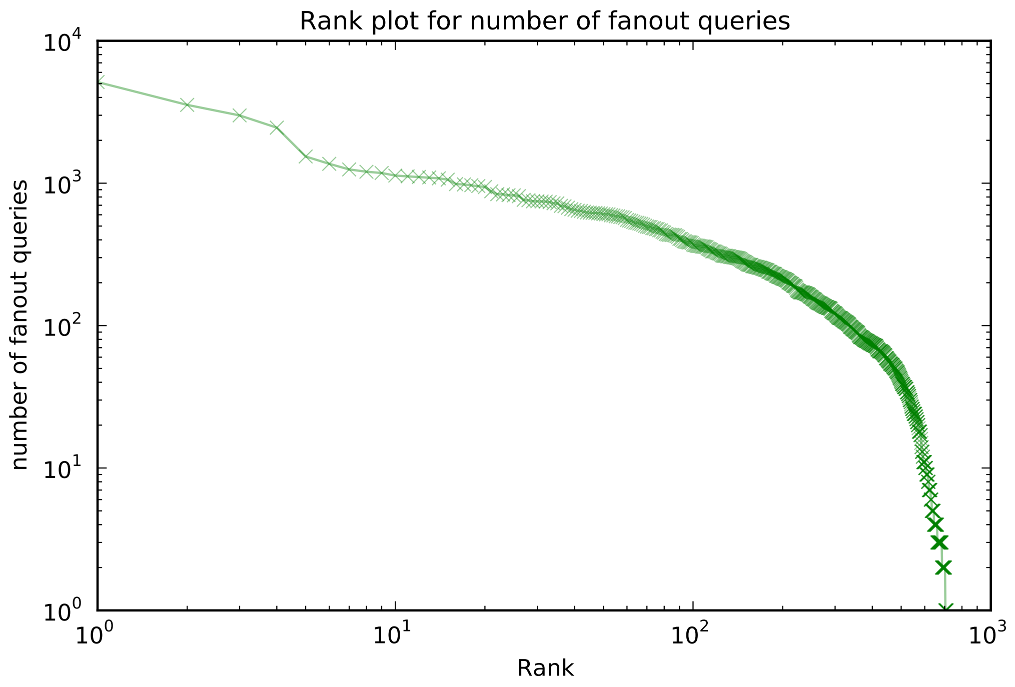
The second question is whether queries are uniformly distributed across users. It is often the case in other systems that requests have certain skew. For example, in a news website it is expected the recent articles are probably requested much more than older ones. In Twitter's case, individual people may be much more active users than others, and some people's profiles may be much more visited than others. Information about query skew is important for optimizations. A large skew makes it harder to balance load across nodes easily, because even if the databases are nominally balanced by user count, not all users contribute to system load the same way. On the other hand, caching may be more effective in large situations than in the case uniform access, which means skew can also be helpful. Either way, a system that assumes a uniform distribution may not be capable of deal with extremes such as particularly popular users or messages. Because of its important effects, also many papers or benchmarks model this kind of skew explicitly by generating queries with some bias, for example [4] and [5].

In order to verify whether this kind of query skew also occurs at Twitter, we aggregated the sample query logs by user and randomly re-sampled about 1000 different users, we got a view of the number of operations they were involved in. The results appear in Figure 2-1.

The plot is on a log-log scale because of the extreme values of the workload. From Figure 2-1 we can see that fanout skew is a lot larger than that for intersection, But in either fanout or intersection, the skew is large. While most vertices are involved in relatively few queries of either type, the most queried vertices are involved in up to



(a) Intersections look linear



(b) Fanouts show different behavior at two ranges

Figure 2-1: Frequency of different queries vs rank

slightly less than 10k fanout queries or up to about 100 intersections (recall this is a small sample of about 1000 vertices that appear in the query logs). Intersection for some reason shows a very straight line, typical of power law distributions, while the fanout frequency somehow is made up of two different straight line segments. These plots show skew results consistent with measurements seen in other workloads, such as Wikipedia [30]. The results for skew on both queries show that there are opportunities in caching, as well as in pursuing which users are the most actively queried and maybe treat them differently. Later in this chapter we check for whether these users also have large degrees, and find a weak but nevertheless positive correlation between these variables.

2.2.3 Query correlations

We saw that some users are queried much more than others. The third question is whether the same users that are popular for intersections are also popular for fanouts. In this section we see there is a weak but unambiguous correlation. The results of these measurements are shown in Figure 2-2.

Interestingly, fanout queries and intersection queries are observably correlated. This suggests that it is meaningful to think of a vertex as being 'popular' for queries as a whole, because being popular for some types of queries implies it is likely also popular for other kinds. Correlation of skews also implies the overall workload skew is more pronounced than if there was no correlation.

2.2.4 Graph data profile

The fourth question was not about the queries to the data store but about the data in it. As of March 2012, Twitter has an active user population of beyond 140 Million [15], and the overall number of vertices is some multiple of that (though most of the load in the system probably comes from those active 140 million users).

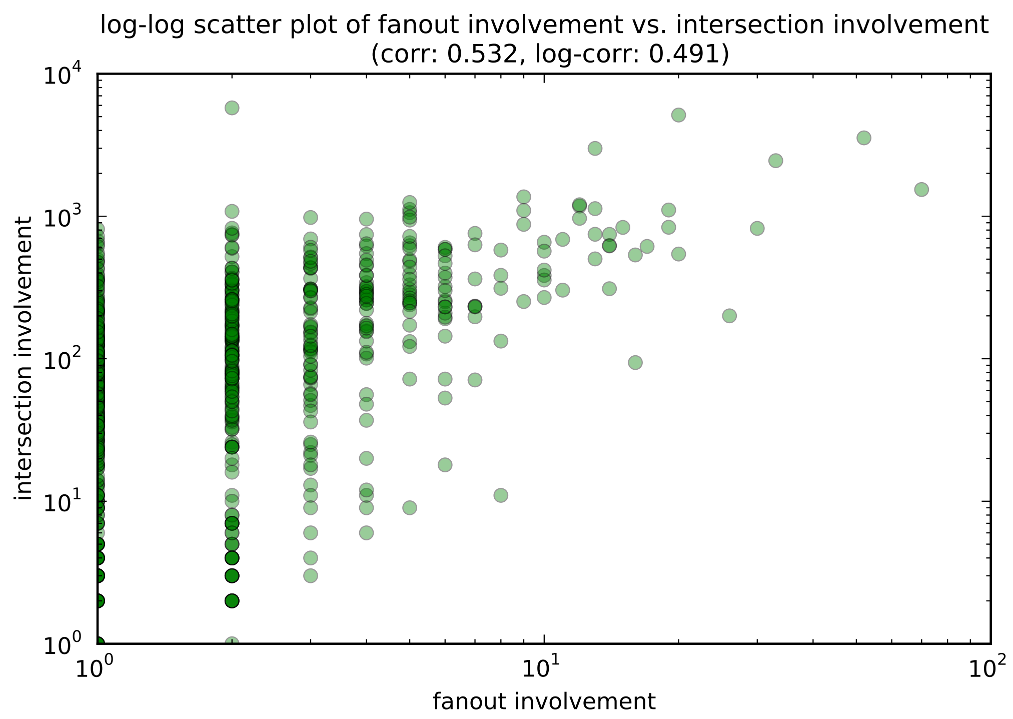


Figure 2-2: log-log scatter plot showing the relation between fanout popularity and intersection popularity

For the experiments I performed, described more extensively in chapter 4 I worked with an older sample of the graph with 130 million vertices and an average follower count of 40, for a total of about 5 billion directed edges. The maximum degree vertex in this sample had about 1 million followers.

The basic Twitter data information is summarized in Table 2.2, side by side with the dataset I used for some of the benchmarks.

quantity	2012 estimates	workload data
number of vertices	$>$ 140 million	130 million
average followers (in-degree)	$<$ 100	40
max number of followers	24 million	1 million
number of edges (using average)	$>$ 10 billion	5 billion

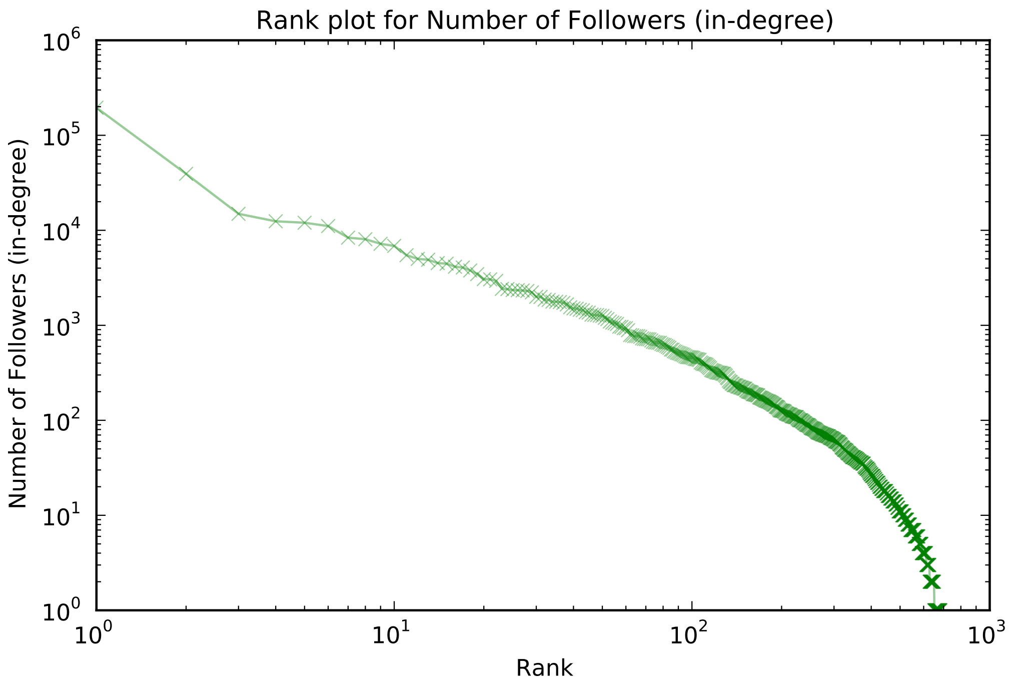
Table 2.2: Basic Twitter data

In the snapshot, the gap between the average in-degree of 40 and the maximum in-degree of 1M is typical of of heavy tailed distributions. In Figure 2-3 A rank plot shows the in-degree as a function of rank. Like with many other naturally occurring graphs, the degree distribution tail is similar to a power law: most of the users have relatively small degree but with a still substantial tail of users having larger degrees. Like with the plots for query skew shown Figure 2-1, the plot looks as a straight line in part of the range, and shows variation along orders of magnitude. From the smoothness of the line We can see how there is a natural progression in degree, which means heavy vertices cannot really be considered outliers because there is a clear progression from the bulk of the data to the extreme values. This suggests that any system needs to know how to deal with both very large and very small degree vertices, and it may need to do so differently. This is one of the motivations behind one optimization approach described in chapter 3.

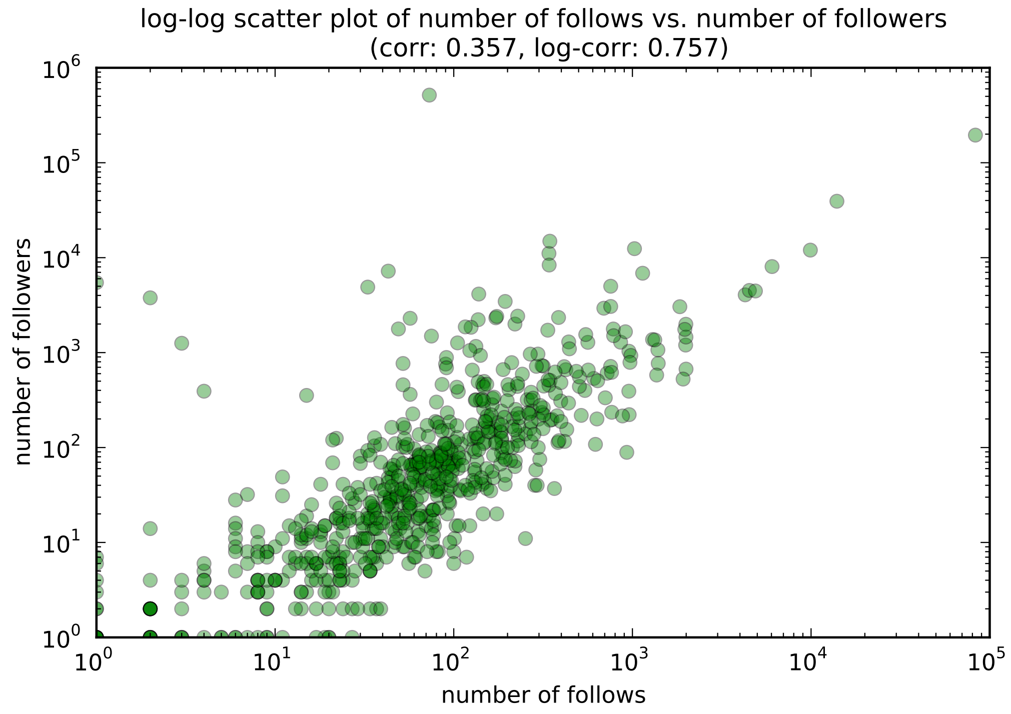
Figure 2-3 again shows the wide range of both the out-degree and in-degree distributions, but additionally also shows how they are very strongly correlated. This correlation holds even the in-degree is decided independently by other users whereas an individual user is in full control of his out-degree. Also note, the out-degree and in-degree are very correlated but an the out-degree is one order of magnitude smaller in range than the in-degree.

The differences between out degree and in degree have suggests interesting optimizations. For example, may be better off storing edge (a, b) as associated with a rather than with b, because this method reduces the incidence of extreme degree cases. Similarly, we could decide users pull Tweets rather than push them. Because pulling is an operation with load proportional to the out degree, whereas pushing is proportional to the in-degree, load may be more balanced this way. I do not pursue these ideas further, but they exemplify the kind of optimizations enabled by knowledge of the specific workload [1].

[1]On the other hand, this pull based approach could increase latency in presenting timelines.



(a) The number of followers (in-degree) of a vertex



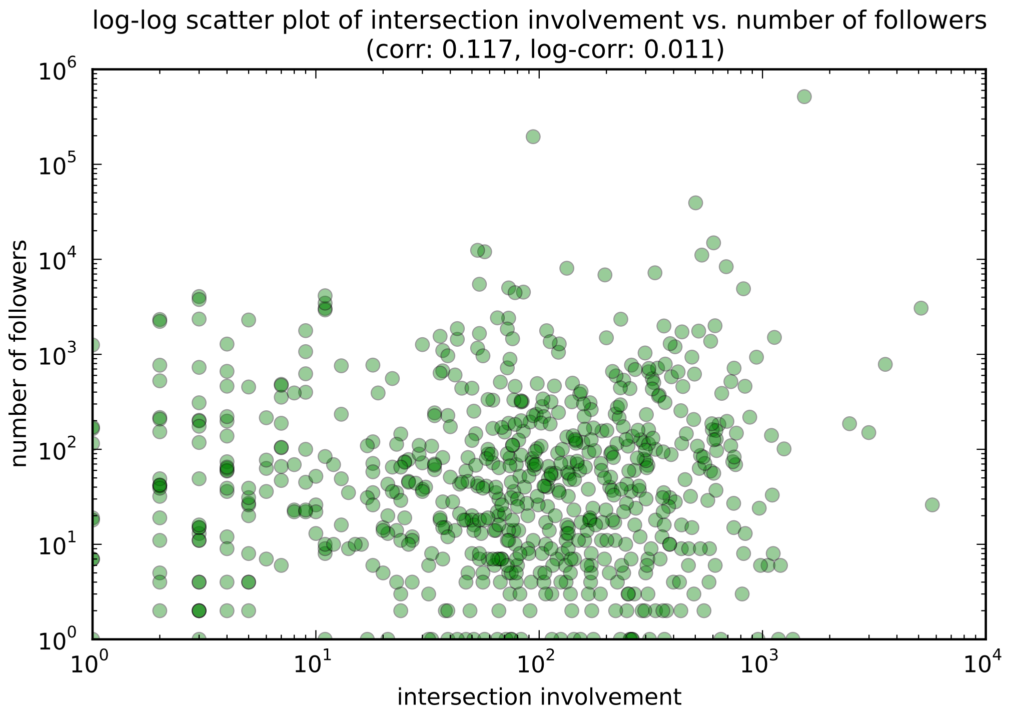
(b) Strong correlation between in-degree and out-degree

Figure 2-3: rank plot of in-degrees over vertices in sample and correlation without-degree

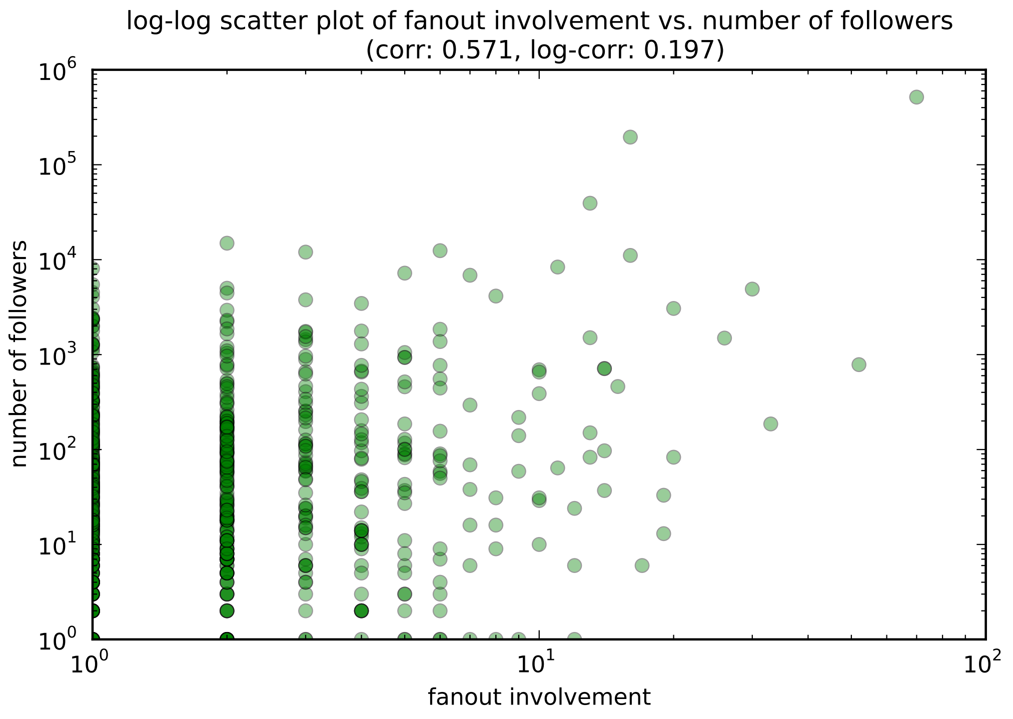
2.2.5 Query-graph correlations

The last question is about correlations between query skew and degree skew. There are arguments why this correlation is plausible: perhaps heavy users are much more active by visiting the site or tweeting more often. Also, a vertex with very large degrees implies a larger historic amount of writes, and would seem more likely to be followed in the future. By joining the log records from section 2.2.1 with the graph snapshot, we checked for any significant patterns. The results are shown in Figure 2-4. The correlations are positive but fairly weak.

In this chapter I introduced the Twitter graph store and its API, including and the two main operations my thesis aims to improve; fanout and intersection queries. I also explained the relevance of these operations from the application's perspective. Most importantly, I presented statistics on actual usage, these included both information about which queries are more common, which turn out to be fanout queries of small page size followed by fanout queries of larger page size, and then by intersection. Also, I exhibited the heavy query and and degree skews in the query patterns and graph respectively, these observations form the basis of the optimization proposals presented in the next chapter.



(a) Intersections seem to be independent from degree



(b) Fanouts are slightly related to degree

Figure 2-4: relation between query popularity and number of followers

Chapter 3

Optimizations

The contributions of this thesis are two different partitioning methods that improve fanout and intersection query performance. Performance refers to many desirable but different properties of a system. As discussed in the introduction, throughput is one such desirable property and it has a direct effect on the costs of growing system. The optimizations aim to improve latency for individual requests. Specifically, the objective of these techniques is to improve service latency, without sacrificing anything at all levels of the distribution, in other words, improving the worst cases without worsening the common case.

There are two good reasons for reasoning about latency in terms of distributions rather than as a single number. The first reason is that if a distribution has a very large spread the mean value is less meaningful. So for instance in the case of Twitter the degree of all users in itself shows high variance, as does the rate at which some users are queried. In situations such as these, average case analysis fails to account for the significant mass concentrated at the extreme cases. A second reason for targeting the full latency distribution rather than only median or average cases is that large latency variability for a service (even with good expected case) creates latency problems when integrating that system into larger ones. The intuitive argument is that when call latency is variable, parallel call latency worsens as we add more calls. In this thesis this same phenomenon affects the design of a partitioning strategy internally later on in this Chapter, in subsection 3.1.2. Perhaps as a result of these reasons it is common practice in service oriented architectures to include latency targets such as bounds on latency at the 99.9 percentile [9].

Another important decision in searching for better partitioning strategies was to exclude adding replication. A few optimization proposals in the literature often involve exploiting not only partitioning but also replication [30]. While those optimizations will likely improve read performance, they involve trading reads off against writes. Writes are one of the core differentiators of the graph store, so there was a conscious effort to focus on improving the reading queries without modifying write costs.

From the information in the previous chapter we know that there are significant degree and query skews. Degree skew causes longer latencies in fanout queries because some vertices simply have very large degrees. The first optimization approach in this chapter reduces the effect of degree skew by making fine grained partitioning decisions for the tail of the degree distribution. The result is a corresponding improvement in the latency distribution. Query skews mean that some queries are more popular than others, this applies also to pairs of vertices involved intersection queries. The second proposed optimization partitions vertices according to how often they are intersected so that they are placed in the same locations, yet, still are spread out so that no only one machine is overloaded. Both of these optimizations are made possible partially because we can make these more fine grained partitioning decisions only of the more significant or active parts of the graph. I expand on the techniques and their motivation in the next sections.

3.1 Optimizing fanout queries

Under the current hash by vertex approach employed in the Twitter graph store, some fanout queries take much longer than others because some vertices have many more edges than others. Because we partition by vertex, then all edges adjacent to the same vertex will always be on the same machines. As new vertices are added to the graph and new edges connect old or new vertices, the differences in vertex degree will become more pronounced. For these two reaons, fanout query latency may become a problem.

Consider that the largest Twitter users have around 25 million followers at the moment of writing. Using conventional hardware parameters, we can estimate the minimum amount of time that different basic tasks take to execute on a vertex of degree 25M. Consider the time it takes to scan sequentially and in memory all of the fanout a heavy user of that degree:

$$25\text{M id} \times \frac{8\text{B}}{\text{id}} \times \frac{250\text{musec}}{\text{MB}} \times \frac{1\text{msec}}{1000\text{musec}} \sim 50\text{msec}$$

So, assuming it takes 250 micro seconds to read 1MB sequentially from memory, it would take 50 milliseconds to complete the task. This number is large enough to fall within the grasp of human perception. As a second example, consider the same task of reading 25M user ids but this time when done over the network or from disk. In that case the calculation changes slightly, and we get

$$25\text{M id} \times \frac{8\text{B}}{\text{id}} \times \frac{10\text{msec}}{\text{MB}} \times \frac{1\text{sec}}{1000\text{ msec}} \sim 2\text{sec}$$

So, to read list of followers sequentially from Disk or to send them sequentially to one node over Gigabit ethernet it takes about 2 seconds. This number is also over optimistic, it assumes we can make use of the full bandwidth, which is not the case if there are any seeks in disk for example.

Admittedly, these two calculations are estimates for the largest of users, but in reality other factors make these calculations fall far below what they actually are. For example, any extra information besides the user ids sent over the wire, or less locality in reads from memory, or paging through the result, would increase the time substantially. From a user standpoint, a difference of less than 5 seconds probably does not detract from the real-time feel of the experience but longer differences could. The graph store is only one of the many components that contribute to the latency between a tweet being posted by a user and the tweet being delivered to the last recipient, but its latency contribution grows with the number of edges in the graph, whereas some other contributors are independent of this latency.

Besides the previous latency argument, there are other reasons to improve fanout performance. There exist several side effects to having some users get Tweets delivered much later than others. One effect is that it causes chains of dependent tweets to stall. Many Tweets are responses or retweets of other tweets. Because delivering out of order conversations would be detrimental to user experience, Twitter implements mechanisms to enforce ordering constraints. One such mechanism is to withhold tweets from delivery until all of their dependencies have been satisfied, or to pull the dependencies upon discovering they are not satisfied. Any unsuccessful attempts to deliver a tweet or repeated calls to pull are a waste of bandwidth. The chances of long chains of stalled tweets increase as latency for large fanouts increases.

Having established the need for methods to improve fanout query latency, we turn to considering methods for improving it. Using parallelism to reduce latency as vertex degrees increase is a natural solution, but the more naïve parallelization approaches do not work well. For example, one possible solution is to partition the edges for every vertex into 2 shards. This technique effectively slows the degree growth rate of the graph. Another possible approach is to instead of hashing by vertex, we hash by edge. This way, all large degree vertices are distributed evenly across all cluster nodes. The problem with this approach is that every query will require communication with all nodes, even queries for small degree vertices that only have a few followers will need to communicate with many nodes. The problem with both the hash by edge technique and hash into two parts technique is that for a great fraction of the vertices, splitting their edges into two separate locations does not improve latency and moreover, needing to query all or many shards is a waste in bandwidth. Many vertices in the graph are actually of small degree, so it is quite likely that the overall system will be slower and have lower capacity than one partitioned using vertex hash.

In order to evaluate what the optimal policy for partitioning edges is in view of the factors of parallelism and communication, I present a simple performance model for fanout queries in the following subsections, and I use it derive the 'Two-Tier' hash partitioning technique, the first of the two partitioning methods proposed in this thesis.

3.1.1 Modelling fanout performance

Intuitively, the main determining factor for fanout query latencies is the number of vertices needing to be read from disk or memory and then sent over the network, this cost is proportional the degree of the queried vertex, which we denote d. If we decide to parallelize, then the amount of work to do per node decreases, but the effects of variability network communication times start becoming comes in as a second determining factor. The more parallel requests there are, the more latency there is in waiting for the last request to finish. Equation 3.1 expresses the tradeoff between parallelism and parallel request latency more formally:

$$L(d, n) = an + max_{i=1}^{n}(L_i + qd/n) \tag{3.1}$$

In Equation 3.1 a and q are system dependent constants, d stands the vertex degree and n for the number of nodes involved in the query. When the n parallel requests get sent to the n nodes, each of those n nodes performs an amount of work proportional to the amount of data it stores, d/n. The initial term an corresponds to a client side overhead in *initiating* the n requests[1]. Additionally, a random delay L_i is added independently to each request, representing latency variability. The total latency $L(d, n)$ is a random variable, because the L_i are assumed to be random as well. Since we are interested in optimizing latency for an arbitrary percentile level p, let percentile(p, X) stand for the pth percentile of a random variable X. For instance, percentile$(0.5, X)$ is the median of a distribution.

3.1.2 The cost of parallel requests

The formula in Equation 3.1 also captures, indirectly, what happens as we increase the n. Whenever we parallelize an execution we potentially reduce the work term d/n done at each node, but add the overhead of making more calls. This overhead has several sources: making each call involves some work, so as we increase the number of calls this work overhead increases linearly with it. Depending on the amount of work we are parallelizing in the first place, this call overhead may become important for large enough n. This overhead is also described elsewhere [12]. A second factor is in the variability of the time it takes to make these function calls. Each call made over a network has a natural variability in latency. Since ultimately we need to wait for the last of the parallel calls to finish, we need to wait for the worst case of n requests. As n increases, the chance of delay also increases.

In this section I explore the sorts of overheads that can occur via simulation. It is reasonable to assume that the effect of making a parallel call to an extra machine will slow the system down a bit, but the exact functional form depends on the variability distribution in the first place. A few simulations show what the effect on latency is like for three different popular distributions: exponential, Pareto (also called a power law) and uniform. As we increase the number of nodes we get the trends in Figure 3-1.

Note that the effects can go from seemingly linear, like in the case of the Pareto distribution, to very sublinear like the case of the exponential. In empirical tests

[1]Though technically, this cost itself may be lowered to $\lg n$ if the initialization of requests is itself done in parallel

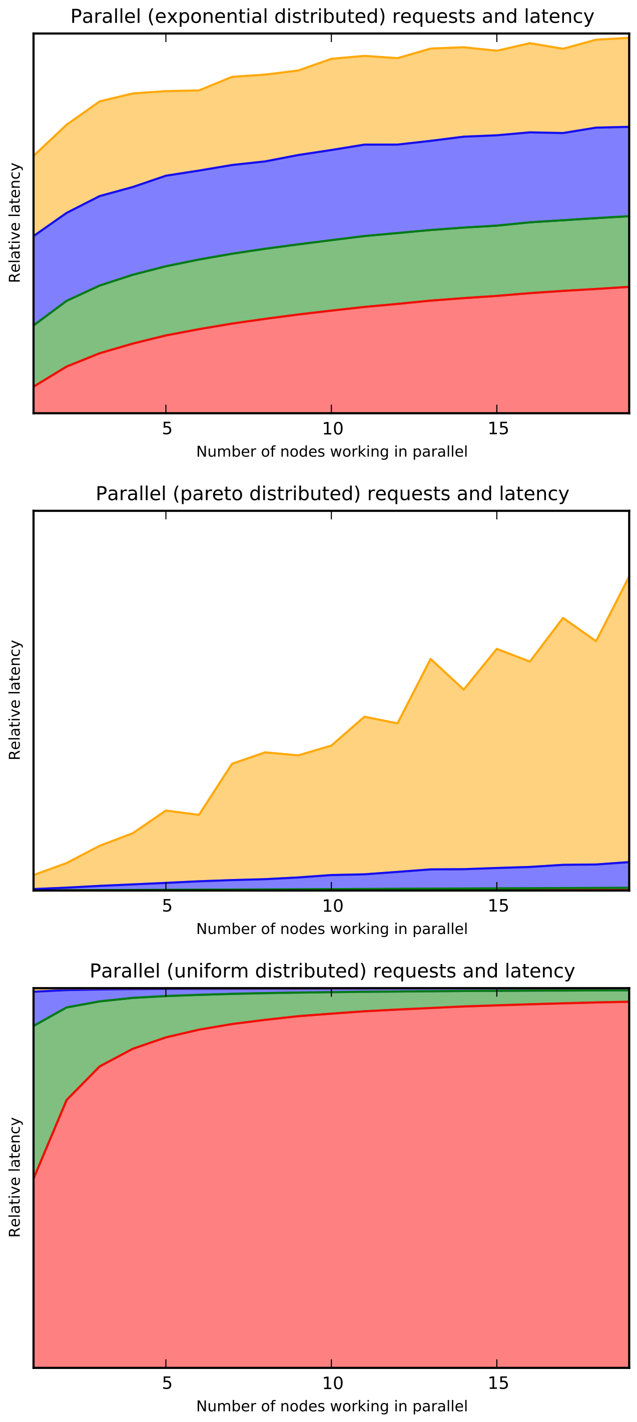


Figure 3-1: Simulated effects of increasing the number of parallel requests on latency for different types of delay distributions (at 50th, 90th, 99th and 99.9th percentiles)

carried out on a real cluster, the linear shape seems to be the more accurate one. Also note that depending on the exact distribution, the different quantiles may diverge completely like in the case of the power law, or may draw closer like in the uniform distribution case.

3.1.3 Optimizing fanout performance

From subsection 3.1.2 we know percentile$(p, max_{i=1}^{n}(L_i))$ for any fixed p is an increasing function of n. We showed that this function may or may not be linear, and the shape depends on the sort of probability distribution. Nevertheless this linearity assumption, also made in [12], helps simplify the calculation about optimal partitioning as follows. We can rewrite the sum $an +$ percentile$(p, max_{i=1}^{n}(L_i))$ as kn for some new parameter k. We then get to Equation 3.2, a simpler model of latency. Also, because the percentile operation does not change the calculation, from now on we avoid writing it explicitly.

$$L(d, n) = kn + qd/n \tag{3.2}$$

Optimizing n to minimize latency from Equation 3.2 is straightforward.

$$\begin{aligned} dL/dn &= k - qd/n^2 & (3.3)\\ \Rightarrow n_{opt}(d) &= \sqrt{qd/k} & (3.4) \end{aligned}$$

The equation shows that the larger q is, that is, the more expensive it is to read objects from memory for instance, the more we will want to parallelize, while also showing that the more expensive it is to make another parallel call, parameter k, the less we want to partition, both statements agree with intuition. From Equation 3.4, the corresponding minimum latency is Equation 3.5. So the optimal latency actually still depends on the vertex degree itself as well as the other system parameters. The equation confirms that the larger either of the cost parameters k or q is, the worse for our system's performance. Also, In a system where there is no cost associated with increased parallelism the 'minimum' latency would be zero, as we could simply increase n linearly with d. This is also reflected in Equation 3.5, which shows $L_{min}(d) \to 0$ as $q \to 0$.

$$L_{min}(d) = 2\sqrt{qkd} \tag{3.5}$$

Also, somewhat surprisingly, the optimal number of nodes into which to divide a vertex grows not linearly with the degree, but as a $\Theta(\sqrt{d})$, as shown in Equation 3.6.

$$\begin{aligned} d/n_{opt}(d) &= d/\sqrt{qd/k} \qquad (3.6) \\ &= \sqrt{dk/q} \qquad (3.7) \end{aligned}$$

One interesting consequence of this calculation is that when we partition every vertex optimally as a function of its degree, the corresponding latency grows as $\Theta(\sqrt{d})$, whereas with a naïve hash by vertex approach, latency grows as $\Theta(d)$. This is an asymptotic improvement. Finally, for a single vertex the optimal number of fanout elements stored per node as a function of a degree is $d/n_{opt}(d) = \sqrt{kd/q}$. Somewhat counter-intuitively, this number is a function of the degree rather than a constant.

Note, this series of results does depend on the functional form we give to cost of increased parallelism. If for example we had modeled it as $k \lg n$ instead of kn, then this results change to the ones in 3.8.

$$\begin{aligned} L(d,n) &= k \lg n + qd/n \qquad (3.8) \\ \Rightarrow dL/dn &= k/n - qd/n^2 \qquad (3.9) \\ \Rightarrow n_{opt}(d) &= qd/k \qquad (3.10) \\ \Rightarrow L_{opt}(d) &= k(\lg(qd/k) + 1) \qquad (3.11) \\ \Rightarrow d/n_{opt}(d) &= k/q \qquad (3.12) \end{aligned}$$

The results from Equation 3.8 are also intriguing, since they seem to challenge intuition a bit less than the ones before, but experiments done in this thesis do show linear cost is not an unreasonable assumption, and also, assuming a linear cost did show improved performance ultimately.

The above analysis were done for an arbitrary but fixed d variable, so the result in Equation 3.4 is valid when we partition a set of vertices of arbitrary but uniform degree. In reality, there are vertices with many different degrees. To optimize for a database with variation of degrees, as happens in reality, one intuitively good but, in our case, suboptimal approach is to redo the analysis above to optimize the average latency, and partition based on it [12]. The solution in that case is Equation 3.15, which follows from Equation 3.13.

The calculation in Equation 3.13 shows this may not be optimal for this workload.

$$\begin{aligned} E_d[L(d,n)] &= E_d[kn + qd/n] && (3.13) \\ &= kn + qE_d[d]/n && (3.14) \\ \Rightarrow n_{\text{avg opt}} &= \sqrt{qd_{\text{avg}}/k} && (3.15) \end{aligned}$$

This result in Equation 3.15 means that for any workload, including Twitter's, the average latency is maximized by optimizing for the average degree. Since the average degree of the Twitter dataset is fairly small at less than 100 followers (see Table 2.2), the result in Equation 3.15 implies $n_{\text{twitter opt}} < \sqrt{100q/k}$. In chapter 4 the values of q and k in my particular test system are empirically found to be 0.01 musec/vertex and 30 musec/node respectively. Incorporating those estimates into the formulae together with the average degree, the optimal amount of partitioning is to put everything in a single node which is equivalent hashing by vertex. And indeed we do find vertex hashing does perform well at the mean. On the other hand, if we wish to improve performance at every level of the distribution (median, 90th percentile, 99th percentile, max), then this averaging technique will not help, which is why it is suboptimal.

A second approach is to split vertices into two classes, ‘light’ and ‘heavy’, and optimize them independently. We can use the result in Equation 3.15 on each of these groups separately. For example, we can divide vertices into those with degree $d <$ 100k and the rest. Because the average degree for the first group can only be smaller than the average for the full graph, the group of ‘light’ vertices will still be partitioned into one single vertex. The heavier group, will be presumably partitioned by at least $\sqrt{100k * 0.01/30} \sim 5$, because the average degree by construction is greater than 100k. This is an improvement, but by the nature of the degree distribution it is likely this group will has an average degree fairly close to the minimum of 100k edges. Hence, the weakness of this this approach is that still fails to deal carefully special care of the largest users. In view of those limitations, we opt to optimize for independently for each vertex. This third approach guarantees an improvement at every tier of the latency distribution. At the same time, this fine-grained approach to partitioning each vertex introduces complications in the system which did not exist for vertex hashing, which we discuss next.

3.1.4 Implementation considerations

By partitioning on a vertex by vertex basis we make optimal global decision, but now we need to store enough information for the the system to be able to split each vertex properly and then route queries to the right storage nodes. For an idea of what the system looks like, refer to Figure 4-1. The hash function approach has the advantage of being stateless, but to support arbitrary partitioning by user id we need to store some lookup table (user ids are arbitrary numbers, with no correlation to any attribute such as location). Fine grained partitioning comes at the cost of complexity in the sharding mechanism. This new state must be stored at every router and updated as the system changes and this state is now also vulnerable to failures, and needs to be recoverable in some way.

One implementation option is to, for each user, keep track of how many parts it is divided into, and then use hashing to figure what the actual locations are. So we only need to keep track of a user id → number of parts mapping, rather than the explicit name of the nodes a user is in. The back of the envelope calculation in Equation 3.16 shows that for a projected graph size of 1 Billion vertices an explicit lookup table of this kind of pair would need a minimum of 10 GB to store. A table this size is small in comparison to other databases, but it may still be too large for coordination and configuration services such as Zookeeper, which in particular expects individual pieces state to be in the range of 1MB [11].

On the other hand, the same calculation shows limiting the lookup table to the 100 thousand largest users would still be possible. So an example of a potential implementation compromise is to only use the optimal strategy on each of the top 100k-1m users. Moreover, because the degree distribution is significantly skewed then techniques like this probably still account for a good fraction of the vertices that most need to be partitioned. This is a case where the skew works in our favor.

$$\begin{aligned} 1 \text{ B (id,num) pair} \times 10 \text{ B/(id,num) pair} &= 10 \text{ GB} && (3.16) \\ \Rightarrow 1 \text{ M (id,num) pair} &\rightarrow 10 \text{ MB} && (3.17) \\ \Rightarrow 100 \text{ K (id,num) pair} &\rightarrow 1 \text{ MB} && (3.18) \end{aligned}$$

Adding or removing edges now has an extra effect, as a vertex gains or loses in degree we may need to change the way we partition it. Some celebrity users make great gains in followers within a few days. For the recovery process, ultimately the routing tables are soft state that can be computed from the data nodes themselves, or we can use a fault tolerant system such as Zookeeper to store the lookup table if it can be made small enough.

Other techniques for how to handle lookup tables are described in [29].

3.2 Optimizing intersection queries

Unlike fanout queries, where the total work done is the same size as the result set, intersection queries often need to do a lot more work than the result set they return: for example, even in the best cases they may need to do linear work only to realize that the intersection is empty. In the worst cases, they may require quadratic work. Also unlike fanout queries, intersection queries may require extra communication steps within the graph store in order to complete.

Intersection queries and their uses were described in chapter 2, and as mentioned in that chapter, one strong motivation for implementing intersections and difference operations within the graph store service itself is to push the filtering as close to the data as possible. Filtering early not only reduces latency but also reduces bandwidth waste, thereby increasing overall system capacity. The optimization I propose takes this reasoning one step further: we prefer the filtering to happen at a single graph store node rather than to have it involve an extra round of communication among internal graph store nodes.

Set intersections are a particular kind of equijoin operation, so many of the same algorithms used for joins are effective to make intersections more efficient. For example, techniques such as hash join, sort-merge join improve intersection from any naive two loop solution, and these kinds of optimizations have already been made within the graph-store. The focus of my proposal here is again in partitioning, to locate vertices across machines in a way that minimizes the number of intersections across nodes. It is hard to know based on the vertices themselves which are likely to be intersected, but this can be predicted by using query logs. In the workload chapter it became clear that a relatively small fraction of users account for many other intersections, so by finding out which users are intersected often, we can likely predict correctly which will continue being intersected. The main ideas for this strategy and the implementation were taken from a similar system that attempts to minimize the amount of distributed transactions in a system [7], in our case, there are no proper transactions but there are distributed queries, which are still relatively expensive. I call this approach workload driven partitioning, and describe it in detail below.

3.2.1 Workload driven partitioning

The goal of this technique is to partition the graph so that often intersected pairs get placed together. To do that, we explicitly partition the graph trying to minimize edge crossings. The important point here is which graph we partition. Many graph partitioner based strategies aim to split the follows graph so as to place adjacent vertices together. This kind of partitioning seems a natural way of handling database partitioning since the graph probably contains clusters defined by external social structure. This social graph driven approach has two drawbacks in the case of Twitter infrastructure. The first is the challenge of partitioning the full graph. Even in the offline case, the graph snapshot was heavy enough that we were not able to get a good result after running METIS for several hours. The second drawback is that the benefits from partitioning the graph explicitly based on communities are not clear either.

This is especially true in the case of Twitter's data services architecture, described in chapter 2, because the user tweet tables and the user timelines are already stored separately, so the only operation happening in the graph store involving multiple vertices are intersections, not tweet deliveries (which would indeed be speeded up if they had to only travel within one machine). Therefore, placing a vertex in the same machine as its followers does not imply more locality in intersection queries unless there is a strong correlation between being followers and intersections. In that case, we might as well explicitly use the workload logs themselves to compute which intersections actually happen often. This is what distinguishes the workload approach from the social graph driven one. The workload driven approach eventually does split the social graph itself and also uses a partitioning heuristic often used to explicitly partition large graphs, but the partitioner itself is based on a different graph, which we call the workload graph. The differences between workload and the follows (social) graph are more clearly explained below.

Definition 3. *Follows graph.*

Directed graph $G = (V, E)$ where the V are users and an edge (v, w) is in E if and only if v follows w.

Definition 4. *Workload graph (for a time interval)*

Symmetric weighted graph $W_G = (V', E')$ with V' a subset of V, with vertex weight for v proportional number of queries involving v in the given interval, and with edges $\{v, w\}$ in E' with weight ω_{vw} proportional to the number of intersections involving v and w in the same time interval.

Both the follows graph G and the corresponding workload graph W_G change over time as edges and vertices are added, but W_G changes even if G remains the same, depending on the queries received in that time interval. Also note that while V' is a subset of V, E' may not be related at all to E. There are other important differences between these graphs. Partitioning G evenly implies that the amount of data stored per node is balanced. Partitioning G minimizing edges means a node is likely to be close to its followers. On the other hand, partitioning W_G evenly implies that actual work done across the partitions is even, and that nodes that are often intersected are close together. Other important differences are that W_G can be small if the time interval is small enough, which provides us a natural way of sampling it by simply sampling the logs. The workload driven approach first computes an explicit partitioning on the workload graph, and then extends it to the rest of the graph via heuristics. The most active part of the graph will be represented in the workload graph, so how optimally partitions the rest of the follows graph is not crucial to performance.

Chapter 4

Experiments and results

4.1 Experimental setup

I implemented a small system to test the different partitioning strategies. The system also allowed to do a other experiments that validated a few of the assumptions upon which the strategies are made, such as the cost of fanouts and work. I describe the system as well as the different experiments done. Experiments include running the query logs on a real graph snapshot, as well as testing them on my own synthetic benchmarks.

The setup, implemented in Java, consists of a DataNode class that stores the actual graph information, and is run on several separate independent server machines, and an APIServer class that translates the graph store API calls `getFanout()`, `getIntersection()` and `getEdge()`, as described in Section 2.1 into several internal remote procedure calls (RPCs) to the separate nodes. The system allows potentially many separate APIServer instances to access all the back-end Data nodes. A diagram for the basic architecture is shown in 4-1.

The APIServer acts as a query router, so in the case of the fine grained strategies it also needs to store lookup tables necessary for routing. The APIServers and the

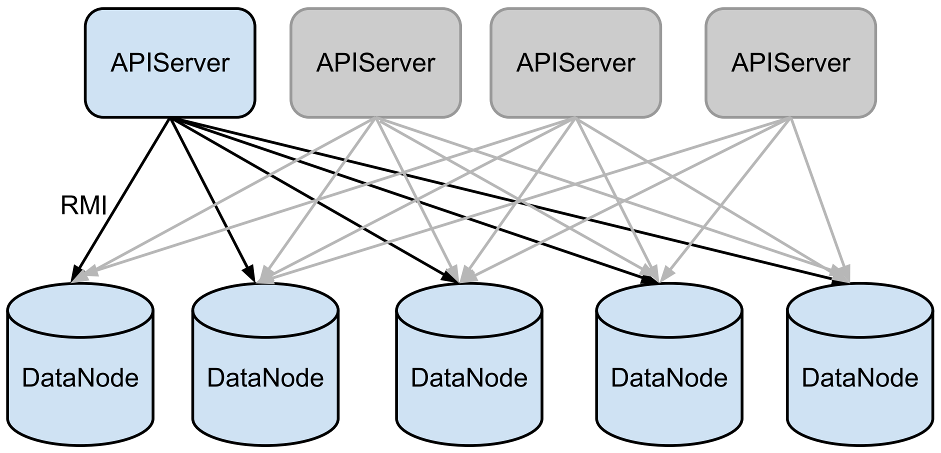


Figure 4-1: Experimental setup, only one APIServer was needed but adding more is possible

DataNodes communicate via Java RMI. Each of the nodes were run on identical hardware and held edge information in memory. For the purposes of this experiment I did not need to support persistence, or concurrent updates since I was measuring read latencies and the effects I was most interested in measuring were the ones due to the amount of data being read. Normally the effects of buffer pools and seeks in disk are much more important, but as more and more services are move to be in-memory, these effects will be less important. Another important effect is the transmission of information across the network, which we address later.

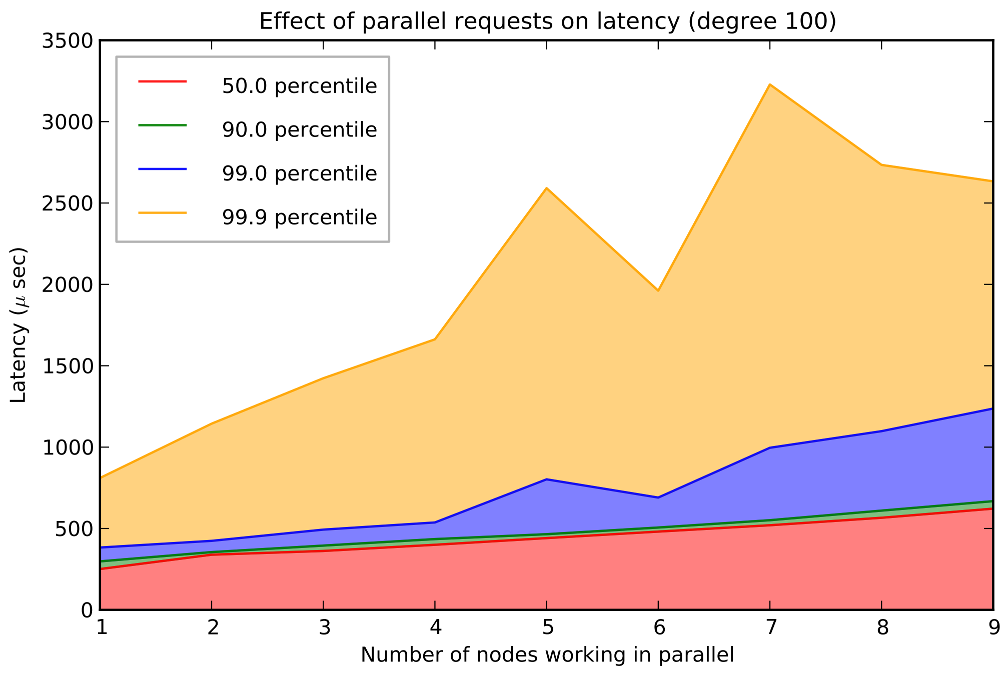
For an in memory databases it isn't as necessary to support concurrent writes because there is no disk IO bottleneck [14]. Finally, in a real system the data can be stored compactly by using custom data structures that take much less space, and using systems implement in languages less memory hungry than Java. But for these experiments, since the data set was static, I used sorted Java arrays to implement a more memory efficient map. This would not be appropriate if there were writes to the database. Using sorted (static) arrays rather than binary search trees keeps the search time for an edge or offset to $O(\lg n)$ but minimizes space devoted to pointers. Ordered reads from an array may perform better than range scans in a more dynamic data structure, since they have a lot more locality, so it could be the effects of our optimizations may be more stronger in a dynamic setting.

Besides the APIServer and DataNode, a Benchmark class and associated scripts were in charge of running the benchmarking workflow: deploying new versions, starting all remote DataNodes, generating and loading data, with parameters specified in a configuration files and gathering statistics. This allowed me to easily evaluate different combinations of parameters such as number of DataNodes, size of the data set, average degree of a vertex, etc. On this setup, I ran the experiments described as follows.

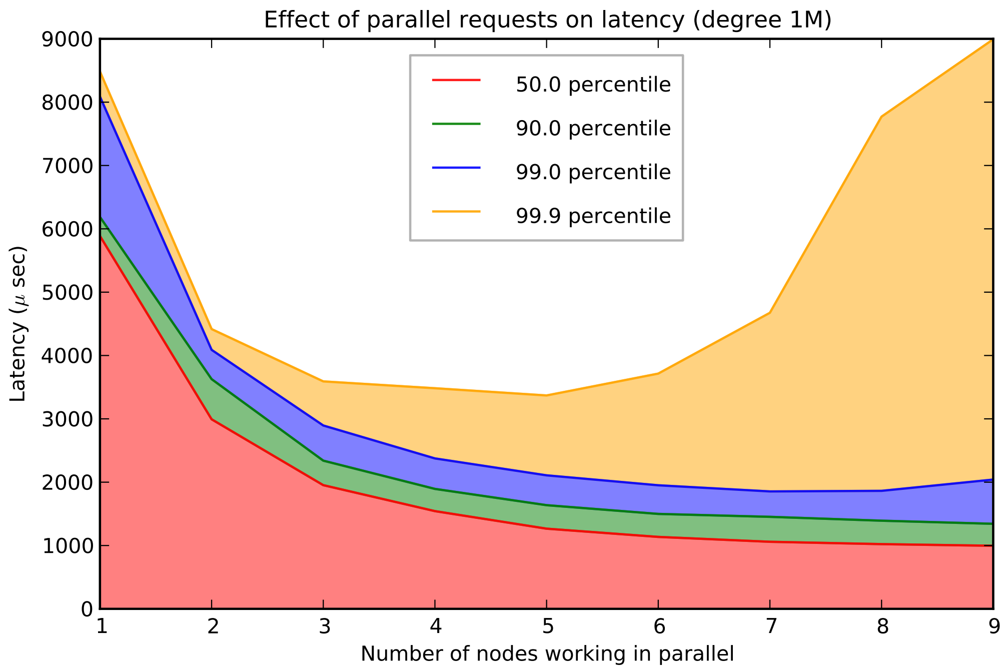
4.2 Fanout performance

The first series of experiments aimed to to determine whether the model of 3.1.1 captured the main factors contributing to latency in a parallelized fanout query. To test this assumption, I carried out a series of experiments varying the degrees of vertices from 100 to 10 million edges, as well as the number of data nodes processing the query. The results are shown in Figure 4-2. In these experiments all vertices in the database had identically large fanouts, and the latency percentiles are computed from a sample of 100 thousand fanout queries.

The plots show how increased parallelism can speed up queries for large degree fanouts, or slow them down for small ones. Additionally, latency was measured at different percentiles of interest. Higher percentiles show more variability, and while we expect even for large fanouts eventually the cost of parallelism will dominate, it seems this effect is felt earlier at the tail of the latency curves. The plots support the model, for fanouts of 100 nodes the effect is a somewhat linear increase in latency,consistent with the kn factor, while for large degree d and low enough n the term qd/n dominates.



(a) for vertices of degree 100



(b) for vertices of degree 1m

Figure 4-2: Effects of increased parallelization on a small degree vertex vs a large degree vertex for a synthetic workload

To achieve these results, I had to remove a network bottleneck and also the bottleneck from putting all the results together at the APIServer. In other words, we cannot reduce overall fanout latency if fanout query goes from being server bound to being either network or client bound. So, obtaining improved performance at the level of the full set of Twitter infrastructure may also require parallelizing the client side.

4.3 Two Tier hashing on synthetic data

4.3.1 For highly skewed synthetic degree graphs

Besides validating the main premise of the Two Tier hash technique, I also used the experiments of Section 4.2 to infer the system parameters k and q from the model using least squares regression. The parallel request cost parameter, k was about 30 musec/nodewhile the work cost q was 0.01 musec/vertex. Using the tuned parameters I was able to choose the optimal amount of partitioning for every vertex in the benchmark graph, each degree d. For this section, the benchmark generates a graph with a degree skew of 1.0 and runs 100k fanout queries from nodes chosen uniformly at random. From the measurements in 2.2.2, we know there is little correlation between a vertex having a large fanout and being queried for more often.

For the first round of experiments I generated a synthetic graph with 1000 vertices and power law (exponent $\alpha = 1.0$) distributed degrees ranging from 1 edge to 10 million, and a skew parameter of 1. The real Twitter graph has a much higher number of vertices, but the actual amount of data should not affect the measurements (on the other hand, repeatedly querying the same single vertex could produce misleading results due to caching effects). Also, the real Twitter graph has a larger α, which should decrease the effectiveness of the Two Tier strategy, so I also experimented with larger α later in the next subsection. On this synthetic dataset, I ran a benchmark of pure fanout requests for vertices chosen uniformly at random, under three different strategies: a simple hash by vertex as control, a two-tier hash, and an all shards also as control, on a cluster of 13 data nodes. Most of the cluster (9 out of 13) nodes were 4 core Intel(R) Xeon(TM) CPU 3.20GHz machines, with cache size 2MB and total RAM 2 GB. The other 4 machines were dual core Intel(R) Pentium(R) 4 CPU 3.06GHz machines, cache size 512 KB and also 2 GB RAM. The differences did not seem to cause significant discrepancies in the metrics.

Our expectation is that for small degree vertices the two tier hash technique should perform better than querying all shards, and about as well as vertex hash. For large degree vertices it should behave more like an all shards partitioning and should be as good as all shards queries, and much better than the vertex hash. In the latency histogram, these observation should translate to the two-tier latency histogram having a shorter tail than vertex hash histogram, and about the same shape as all shards. It should also imply that the lower percentiles (20th percentile for example) are better for vertex hash as well as two-tier than for all shards. The full histograms are plotted in Figure 4-4 and the main results summarized in Table 4.1.

As we expected, at higher percentiles (90th, 99th and 99.9th) the Two Tier and All Shards perform comparably, while the vertex hash is substantially slower (by factors of more than 2). As expected, this improvement of two-tier hashing over vertex hashing is even higher at the 99th percentile: all-shards and two-tier each take about 9000 musec, whereas single vertex took 76000musec. The median latencies (and below) for vertex hashing and two tier are similar, whereas All shards does not do very well at these ranges. Lower latency percentiles show more strikingly how all shards queries are expensive for smaller nodes.

strategy	5th	10th	30th	50th	90th	99th	99.9th
vertex hash	546	590	647	748	9938	75884	149500
two tier	299	348	452	549	2605	8938	48156
all shards	824	850	919	998	2663	9175	50628

Table 4.1: Comparison of latencies at different percentiles (in micro seconds)

Note, the relative improvement from using the two tier strategy depended both

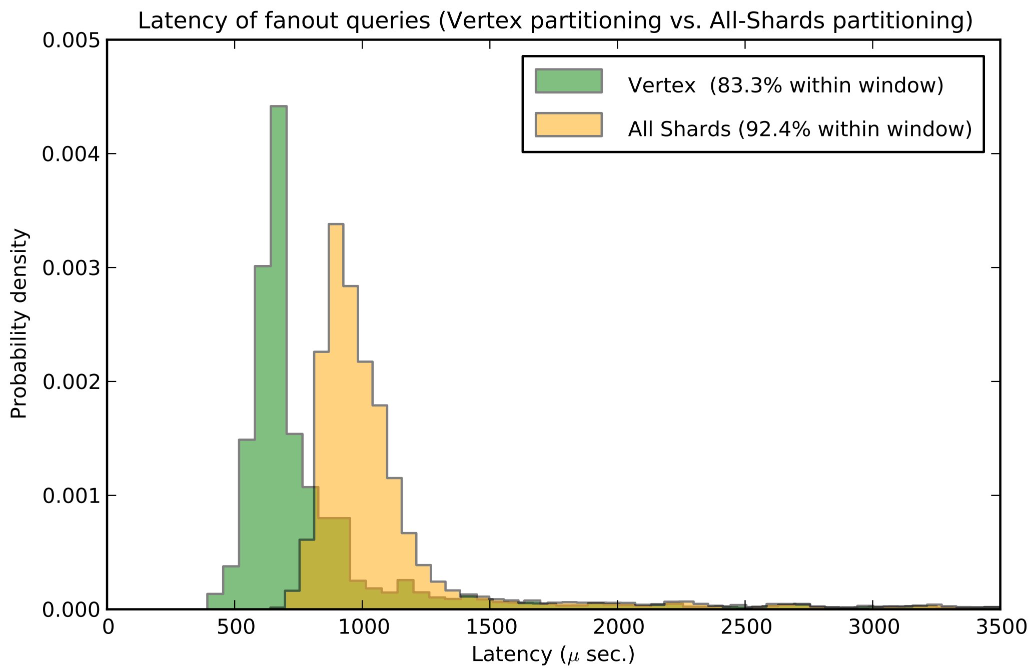
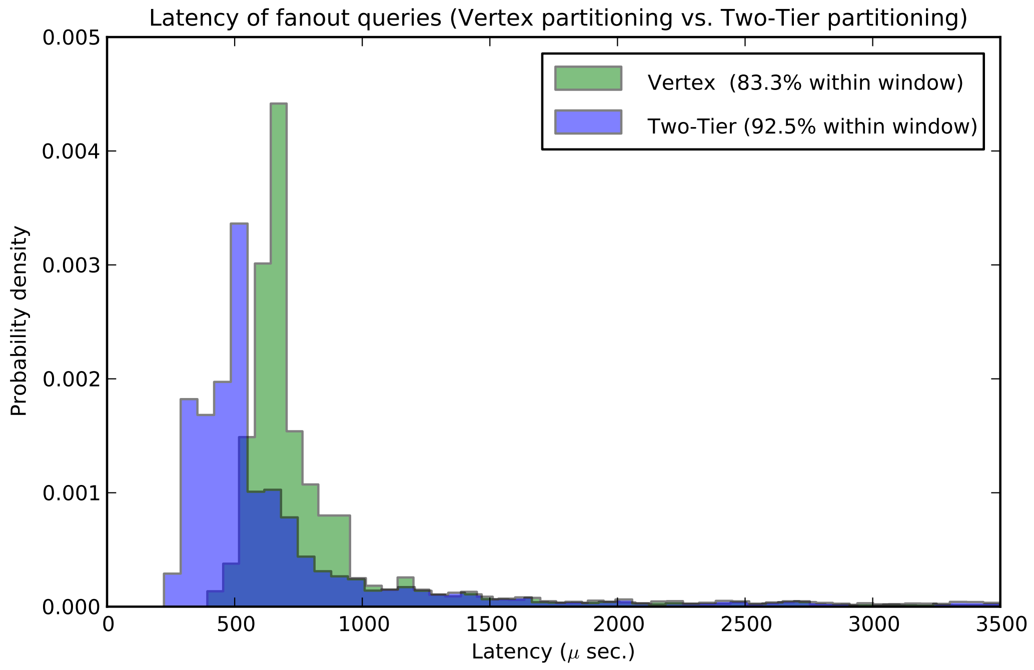


Figure 4-3: The two control strategies: vertex sharding and all-shards partitioning. Note the substantial fractions of data lying beyond the window range. See also Figure 4-4

on tuning it and on the skew of the data. Here the skew parameter was 1. The larger the chances of extreme cases, the better a two-tier strategy performs relative to either strategy at all latency percentile ranges.

4.4 Two tier hashing on real data

I also tested the strategy on the graph snapshot, using the fanouts from the query logs. The dataset used is described in Table 2.2. Figure 4-5 shows the difference in latency between Vertex hash partitioning and two-tier partitioning for the real workload. A result summary is shown in Table 4.2. The hardware setup for this section was different than the one for Subsection 4.3.1, the machines had 80 GB of RAM, and I only used 5 of them, and the graph has less skew than the one in that section as well. Also, at that moment I had not yet implemented fine grained partitioning, but simply chose at threshold from which to partition. The tail of the



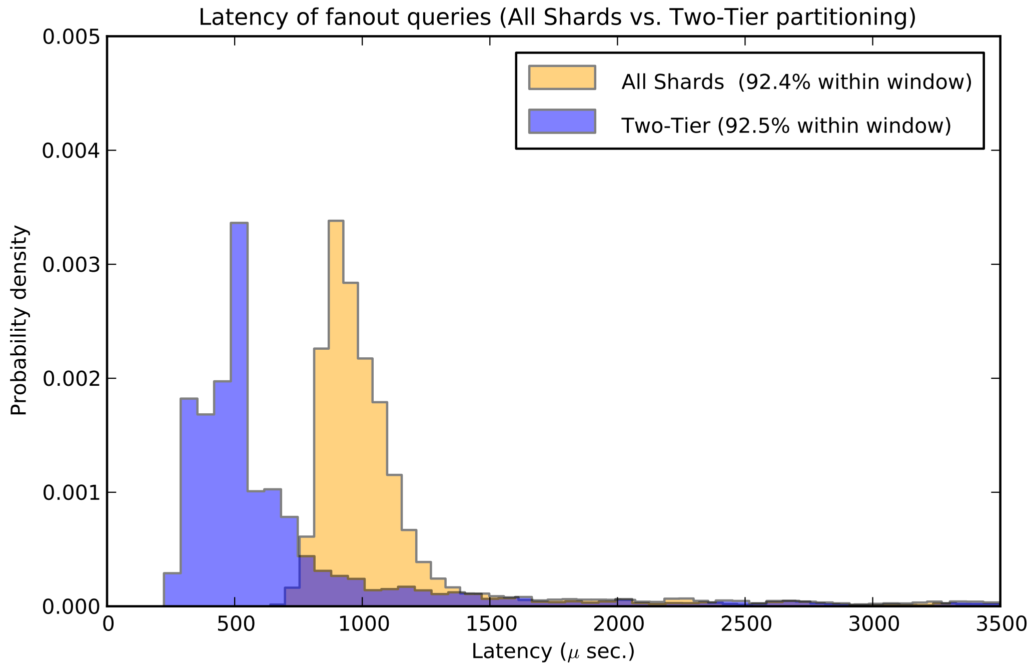


Figure 4-4: Comparing the two-tier strategy and the two control strategies on fanout latencies for synthetic graph and queries on the same synthetic workload as Figure 4-3

distribution is reduced substantially, and the two-tier sharding histogram shows a bimodal distribution. One possible explanation for the the bimodal shape is that I partitioned all nodes after a threshold equally, rather than in the more careful way explained in the previous section. The benefits are still clear from the table.

strategy	50th	90th	99th	99.9th
vertex hash	468	743	1113	2030
two tier	376	548	947	1756

Table 4.2: Comparison of fanout latencies at different percentiles on real data (musec)

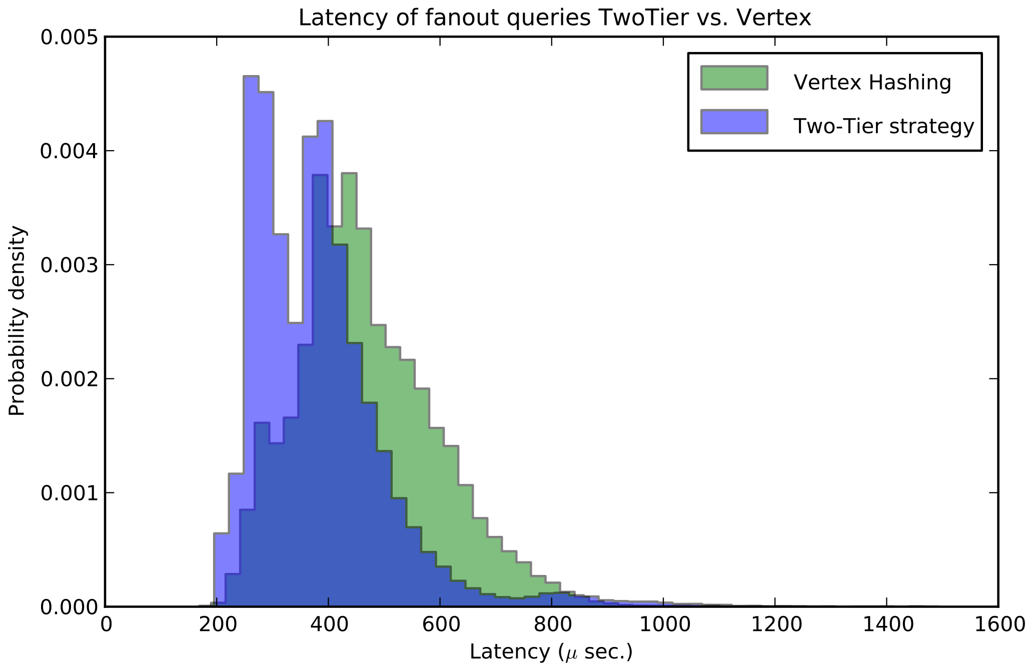


Figure 4-5: Comparison of fanout query latencies of Two Tier and Vertex hash partitionings using real graph and logs

4.5 Intersection queries on real data

Similarly, I tested the workload driven partitioning strategy on the graph snapshot described also using the the query logs. For this particular experiment, first it was necessary to generate the static partitioning, using the log records themselves, the graph snapshot and the METIS partitioner.

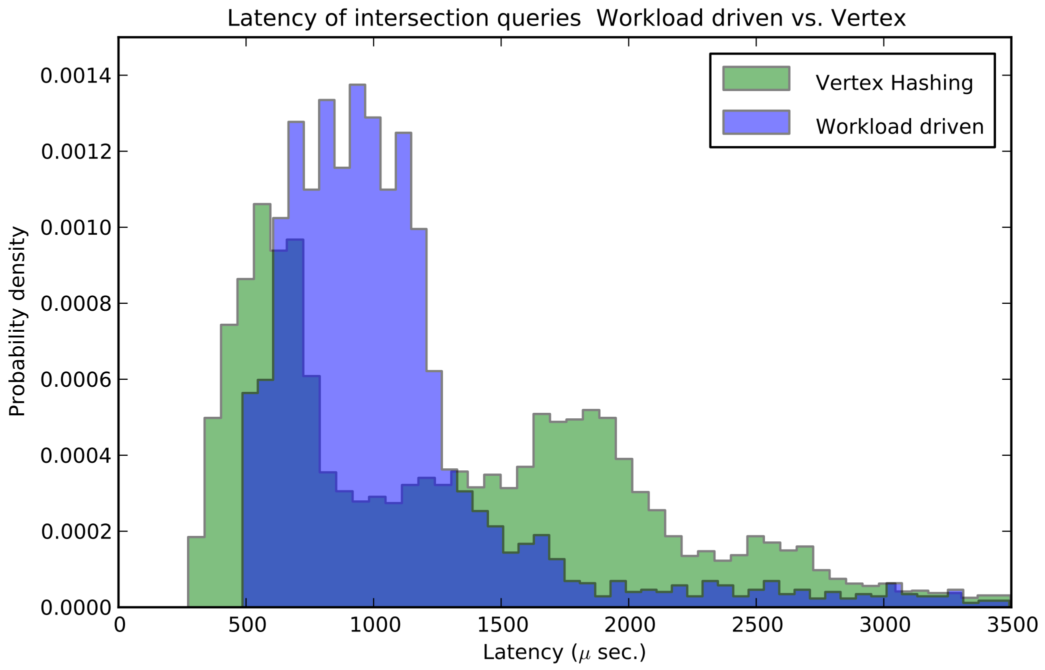


Figure 4-6: Comparison of intersection query latencies of Workload driven and Vertex hash partitionings using real graph and logs

Since the query graph includes only a about 10% of the total nodes, we partition the rest of the graph by using a vertex hashing function and similar to described in Section 3.1. We replayed traffic from the same log on a running system loaded with the snapshot from the previous section. One possible effect of this is that performance is better than it would really be on previously unseen data, but the experiment is still meaningful as an upper bound in performance. Further experiments are needed in order to evaluate how the strategy performs on workloads less related to the training data. Resulting histograms for running these intersection queries are shown in Figure 4-6, and Table 4.3 shows the comparison in latency.

strategy	50th	90th	99th
vertex hash	1608	10562	1038754
workload driven	975	1838	12010

Table 4.3: Comparison of intersection latencies at different percentiles on real data (musec)

Chapter 5

Related work

In this thesis I presented a particular approach to partitioning tailored specifically for improving fanout and intersection queries in social network type data. Many other papers have presented proposals for different kinds of partitioning strategies.

Dewitt et al. [12] show a partition technique that optimizes parallelism for the average query, for a general database. They propose a model very similar to the one used to evaluate two tier hashing in this thesis, but are thinking of general workloads. They propose a Hybrid Range partitioning strategy, where they split the keyspace into ranges and then use hashing above that. They show improvements on both latency and throughput when compared to either pure hashing or pure range partitioning. Also in the general database partitioning area, Schism [7] partitions tables explicitly to minimize the number of distributed transactions. It does this based on past workload data,and by modelling the problem as a graph partitioning problem. They compute the partitioning by employing the METIS algorithm.

On a different line of work, there are systems as Feeding Frenzy [26] have also exploited the differences in workloads across Twitter users, and leverage it to propose a hybrid push-pull strategy for Tweet delivery. They proposed making a pair by pair decision on whether to push Tweets at publication time or pull them later. The technique improves performance because some users publish much more rapidly than others consume. By not pushing Tweets that never get seen by a slow consumer the system can save bandwidth.Also in the same line of making fine grained partitioning decisions on social network graphs, Graphlab [1] also uses optimizations such as treating large degree vertices differently than small degree vertices to enable more parallelism.

A different part of the literature aims to compute graph partitions. The METIS [25] heuristic has a reputation for producing good partitions and is used prior to work by parallel processors to partition graph like objects such as image meshes, a problem of allocation similar to distributing data in a database. METIS has proven so successful that systems like Schism use it in a similar way to how it was used in this thesis. On the other hand, METIS suffers from being an offline heuristic, so there are proposal for feasible online graph partitioning algorithms, such as the one by Stanton and Kliot [28]. Some other approaches to graph partitioning are explicitly designed for social networks and attempt to uncover community structure in order to find natural partitions of graphs. However, these approaches do not aim to produce necessarily balanced partitions.

Other papers propose systems to explicitly support Twitter like workloads. Pujol, Erramilli et al [23] propose an online heuristic called SPAR (for social partitioning and replication) for partitioning a graph such as Twitters that is constantly being written to. Their goal is for the system to minimize the number of graph edges going across partitions. This system reacts to each node or edge addition and deletion using a local, greedy heuristic. Unlike the approach in Schism, SPAR partitions happen online. Also unlike Schism, SPAR partitions are based on the social graph itself rather than based on the workload. For operations considered in this thesis such as intersection queries, it is unclear that there is a correlation between social graph edges and workload graph edges. Authors of both Pregel and Giraph, two different graph processing systems, point out that careful data graph based partitioning in their systems may not work well because in practice inter vertex communication in the workload is not necessarily related to the edges in the data graph [22] [3].

A related issue that arises when creating custom partitions is to be able to store it. Schism attempts to solve this problem by using decision tress to infer simple predicates that explain the graph partitioning. Pujol et al, in their test implementation of SPAR, keep an explicit lookup table and scale it by implementing it as a DHT. Tatarowicz et al [29] explain different methods to help solve the problem of storing state for fine grained partitioning strategies. Several techniques that they propose are used in this thesis, such as keeping track only of a small subset of the vertices and using a hash partitioner for the rest.

Conclusion

I have presented two data partitioning strategies for improving query latency in Twitter's graph data store. The first one, two-tier Hashing, improves fanout query latency across the whole distribution of latency for fanout queries. The second strategy, workload aware partitioning, improves intersections latency.

Two-Tier hashing treats large degree vertices differently from the average degree vertices. It spreads heavier vertices across more machines than lighter vertices, this enables more parallelism for large fanout queries, while it also preserves locality for small fanout queries. The exact number of partitions a used to spread the edges for a given vertex can be computed using an explicit performance model, and is a function of vertex degree and two other system dependent parameters. The strategy requires the partitioning system to be aware of the different vertex degrees and parameters in order to route queries correctly.

Because keeping explicit information for every vertex of the graph would be prohibitive, the implementation in this thesis divides vertices into two tiers. The partitioner only needs to keep track of which vertices are in the top tier, and for each of those it keeps degree information. For the rest of the vertices it uses a default hash by vertex partitioner. By keeping detailed degree information for the heaviest vertices it can optimize for each individually. Moreover, Because most light vertices should not be partitioned into multiple pieces anyway, the default vertex hashing will be optimal for them too. For this reason, this two tier approach performs just as well as the best strategy at different levels of the degree distribution. Experiments using both synthetic and real data workloads provided by Twitter show two-tier hashing is keeps latency lower than vertex hashing.

Workload aware partitioning attempts to place vertices that are often intersected together into the same machine. Intersection operations that occur across machines are much more resource intensive than operations occurring within a single machine, so by minimizing cross-machine intersections, we improve both intersection latency and reduce usage of external bandwidth. Workload aware partitioning can find good partitions because it uses actual query logs and graph partitioning software to calculate which vertices should be located together.

Like with two-tier hashing, workload aware partitioning must keep track of explicit partitioning decisions for each vertex. Because keeping track of all vertices in a graph the size of Twitter's would be difficult, this implementation only keeps track of the more actively accessed vertices, and uses a default hash function on the rest. Because there is a large skew in the query distribution, keeping track of the most queried vertices still allows workload aware partitioning to perform substantially better than vertex hash partitioning.

Bibliography

[1] D Bickson. Preview of graphlab v2 new features! Blog post, December 2011. http://bickson.blogspot.com/2011/12/preview-for-graphlab-v2-new-features.html.

[2] L Breslau, P Cao, L Fan, G Phillips, and S Shenker. Web caching and zipf-like distributions: Evidence and implications. In *in INFOCOM 99: Proceedings of the 18th IEEE International Conference on Computer Communications*, pages 126–134, 1999.

[3] Avery Ching. Giraph: Large-scale graph processing on hadoop. Talk at Hadoop Summit, 2011. http://www.youtube.com/watch?v=l4nQjAG6fac.

[4] B F Cooper, R Ramakrishnan, U Srivastava, A Silberstein, P Bohannon, H-A Jacobsen, N Puz, D Weaver, and R Yerneni. Pnuts: Yahoo!s hosted data serving platform. In *In Proceedings of the International Conference on Very Large Databases (VLDB*, 2008.

[5] B F Cooper, A Silberstein, E Tam, R Ramakrishnan, and R Sears. Benchmarking cloud serving systems with ycsb. In *In SoCC: ACM Symposium on Cloud Computing*, 2010.

[6] Oracle Corporation. Partitioning with oracle database 11g release 2. White Paper, September 2009.

[7] C Curino, Y Zhang, E Zones, and S Madden. Schism: a workload-driven approach to database replication and partitioning. In *In Proc. VLDB*, 2010.

[8] Jeffrey Dean. Achieving rapid response times in large online services. Talk Slides, March 2012. http://research.google.com/people/jeff/latency.html.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[10] Lauren Dugan. Twitter users send 32,000 tweets-per-minute during champions league final. mediabistro, May 23 2012. http://www.mediabistro.com/alltwitter/new-tweets-per-second-record_b22987.

[11] Apache Software Foundation. Developer documentation, 2012. http://zookeeper.apache.org/doc/current/zookeeperProgrammers.html.

[12] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 481–492. Morgan Kaufmann, 1990.

[13] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[14] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1376616.1376713.

[15] Twitter Inc. One hundred million voices. Company blog post, September 2011. http://blog.twitter.com/2011/09/one-hundred-million-voices.html.

[16] Twitter Inc. Cassovary big graph processing library. Company blog post, 03 2012. http://engineering.twitter.com/2012/03/cassovary-big-graph-processing-library.html.

[17] Twitter Inc. Flockdb. Source code repository, 2012. https://github.com/twitter/flockdb.

[18] Twitter Inc. Twitter turns six. Company blog post, March 21 2012. http://blog.twitter.com/2012/03/twitter-turns-six.html.

[19] B W Kernighan and S Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[20] D Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

[21] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[23] J Pujol, V Erramilli, G Siganos, X Yang, N Laoutaris, P Chhabra, and P Rodriguez. The little engine(s) that could: Scaling online social networks.

[24] Marko A. Rodriguez and Peter Neubauer. The graph traversal pattern. *CoRR*, abs/1004.1001, 2010.

[25] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[26] A Silberstein, J Terrace, B Cooper, and R Ramakrishnan. Feeding frenzy: Selectively materializing users' event feeds.

[27] Steven Skiena. *The Algorithm Design Manual (2. ed.)*. Springer, 2008.

[28] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. Technical Report MSR-TR-2011-121, Microsoft Research.

[29] A Tatarowicz, C Curino, E Jones, and S Madden. Lookup tables: Fine-grained partitioning for distributed databases.

[30] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[31] W3C. Sparql query language for rdf. W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query/.