Moolle: Fan-out Control for Scalable Distributed Data Stores

SungJu Cho*, Andrew Carter*, Joshua Ehrlich† and Jane Alam Jan*
*LinkedIn Corp
2029 Stierlin Ct. Mountain View, California 94043-4655
Email: sungjuc73@gmail.com, acarter@linkedin.com, jan876@gmail.com
†WePay, Inc.
350 Convention Way Redwood City, California 94063
Email: ehrlichjoshua@gmail.com

***Abstract*—Many Online Social Networks horizontally partition data across data stores. This allows the addition of server nodes to increase capacity and throughput. For single key lookup queries such as computing a member's 1^{st} degree connections, clients need to generate only one request to one data store. However, for multi key lookup queries such as computing a 2^{nd} degree network, clients need to generate multiple requests to multiple data stores. The number of requests to fulfill the multi key lookup queries grows in relation to the number of partitions. Increasing the number of server nodes in order to increase capacity also increases the number of requests between the client and data stores. This may increase the latency of the query response time because of network congestion, tail-latency, and CPU bounding. Replication based partitioning strategies can reduce the number of requests in the multi key lookup queries. However, reducing the number of requests in a query can degrade the performance of certain queries where processing, computing, and filtering can be done by the data stores. A better system would provide the capability of controlling the number of requests in a query. This paper presents Moolle, a system of controlling the number of requests in queries to scalable distributed data stores. Moolle has been implemented in the LinkedIn distributed graph service that serves hundreds of thousands of social graph traversal queries per second. We believe that Moolle can be applied to other distributed systems that handle distributed data processing with a high volume of variable-sized requests.**

I. Introduction

A. Background

Large-scale online data services in industry have widely adopted distributed data systems as a way to deliver fast response time, scalability, and fault tolerance. Systems such as Memcached[6] have shown that distributed in-memory key-value stores have lower latency for single key lookup queries than monolithic disk based stores.

The most popular data partitioning strategy in distributed data systems is hash based horizontal partitioning because it is easy to scale up and simple to manage. The pseudo-random hash function used in the partitioning achieves even data distribution across multiple data stores and the location of data can be determined without mapping table or routing information. New data is simple to add because its location is already determined by the hash function. The hash based horizontally partitioned distributed data service can process a single key lookup query efficiently because a single partition can process the query. For example, to process a member's 1^{st} degree (hop) connections lookup query, the query processor fetches the connection data from a single partition.

Processing multi key lookup queries under the horizontal partitioning is not as efficient as single key lookup queries because the query typically requires fetching data from multiple partitions. For example, consider a multi key look up query that fetches a member's 2^{nd} degree connections, the union of the member's 1^{st} degree connections' 1^{st} degree connections. In the 2^{nd} degree connections look up query, data from many partitions is needed because the member's 1^{st} degree connections are randomly distributed. For users with large 1^{st} degree connections, there is a high probability that data from all partitions will be needed. The large number of fan-out requests in the query can cause issues such as network congestion, tail-latency, and CPU-bounding.

Tail-latency[5] refers to the problem that occurs when the processing time of a query issuing a large number of fan-out requests is dominated by the longest response time among the fan-out requests. Among the participating partitions, even one poorly performing partition is enough to cause a significant increase in the query response time. This issue is difficult to resolve but common in distributed data services in industry. In most production data store nodes, each data store contains various additional processes for maintenance and monitoring purpose. Those supplementary processes may temporarily have a negative impact on the query performance. For example, a log transfer process can block the query processing work by disturbing operating system's file system cache whenever it deletes large log files after transferring. Garbage collection activities in JVM applications can cause sudden increases of response time for the data store partition that are hard to control or predict. The larger the number of fan-out requests, the higher the probability of requesting from a partition with temporary slow performance.

CPU-bounding is also known as Multiget Hole[8] problem. When the overhead of processing the network stack is comparable (or dwarfs) the processing costs of the query, a large number of fan-out requests can increase the load on the CPU. Adding more data-partitions to scale horizontally increases the required CPU power per query by increasing the number of fan-out requests in a single query. If the CPU-bounding issue starts, requests and responses will wait in a processing queue; this will significantly increase the processing time of most queries.

Past research has attempted to address these issues of horizontal partitioning by reducing the number of fan-out requests. One notable direction is a fine-grained partitioning[2], [4], [12] scheme that generates partitions optimizing data locality. High data locality enables the multi key lookup query processing to minimize the number of fan-out requests using just a single partition or a small number of partitions. For example, a partitioning using a community detection algorithm will partition a member's data with high data locality such that the data for a member's 2^{nd} degree network would be stored on a single partition. In this case the fan-out problem disappears as a single node can serve the request. Due to the huge computational overhead of construction and maintenance, the fine-grained partitioning has not been widely adopted to the ever-changing data such as Online Social Networks.

Another notable solution to the horizontal partitioning issues is to minimize the number of partitions participating in a query processing using the set cover based algorithm and data replication[14], [16]. From the data replication, each partition contains different set of data such that the set cover based routing algorithm effectively selects the minimal set of partitions. The approach effectively solves the scalability and tail-latency issues by reducing the amount of fan-out using a relatively simple partitioning strategy. Because the minimal set of partitions computation is a NP complete problem[7], most practical implementations use the greedy set cover approximation. The greedy nature of the routing algorithm makes it difficult to control the number of the participating data stores. It always attempts to select the minimal set of data stores for the set of partitions. This limitation is not suitable to process some queries that require more participating data stores in order to distribute computational and processing load.

We believe that distributed data services should have the ability to control the number of fan-out requests in a query processing because industry applications need to handle a diverse workload. The query processor should be able to selectively determine and control the number of data store nodes based on each query's processing characteristics and global performance related statistics. It should be able to select the minimal set of partitions to process small multi key lookup queries, and also select maximal set of data store nodes to process CPU intensive queries. Furthermore if all the data store nodes have a high number requests, the query processor should be able to reduce the number of requests for incoming queries. Thus, we are motivated to design a system that can handle these types of queries by controlling the number of fan-out requests per query basis.

B. Terminology

In this paper we will use following terminology:

- Partition: a division of data divided by a partitioning strategy
- Data store node: a physical server machine. One data store node can have multiple partitions.
- Client node: the physical machine handling incoming queries and dispatching requests to data store nodes.
- Query: request issued by applications and handled by the client.
- Request: the request issued by the client and handled by the data store node.
- Query processor: application that handles incoming queries. Both client and data store nodes have query processors.
- Fan-out requests: the requests issued by a client node to process an incoming query.
- Fan-out control: a process to determine the number of fan-out requests.
- Routing table: contains the key/partition and partition/data store node mapping information.
- Queries Per Second (QPS): the number of incoming queries to a client node per second.
- Requests Per Second (RPS): the number of dispatching requests from a client node per second.

C. Our Contributions

In this paper, we present Moolle, a per query basis data processing system that can control the number of data store nodes that participate in the processing of a query while using a simple hash-based horizontal partitioning and replication. Moolle consists of replicated data store nodes and a client side load balancer. The load balancer can limit the number of data stores participating in the process of an incoming query using a limiting parameter provided by query processing applications. This allows the query processor to find a better balance between reducing server side computation and reducing client side computation. Moolle provides API options to specify the degree of fan-out requests to query processor.

While Moolle is not an intelligent query processing system that determines an optimal fan-out strategy in distributed data services, it allows programmers to implement the intelligent query processors. The query processors utilize a knob to control fan-out requests on a per query basis, which allows applications to optimize the entire system to meet its objectives such as per query performance and global data store stability optimization.

We believe that Moolle is the first general-purpose distributed data system that provides the ability to control the fan-out of requests on a per query basis while considering the global state of the distributed data system. We have implemented this system as part of the LinkedIn's Norbert [11] open source project. We are using this system in the production distributed graph service at LinkedIn. With Moolle controlling the fan-out of requests between clients and data store nodes in our system we have decreased the average and 99th percentile response time of requests between client's node and data store nodes by around 40%. Additionally, the graph service can limit the requests per second (RPS) based on the growth of the incoming queries per second (QPS) to avoid increasing queuing time, which enhances the scalability of system by preventing CPU-bounding. While this paper focuses on Online Social Networks and our Distribute Graph Service, we believe that other systems would benefit from the same fan-out control mechanisms.

In Section II, we detail LinkedIn's distributed data service architecture, which inspired Moolle design. In Section III, we discuss the important design aspects such as partitioning/replication strategy, fan-out control load balancer and fan-out controller. In Section IV, we present the analysis of the latency, QPS and RPS in the distributed query processing. These are important aspects that influenced our Moolle based fan-out controller designs. In Section V, we highlight the important aspects of Moolle based on simulation results, and summarized the contributions to the LinkedIn production system. In Section VI, we discuss related works.

II. LinkedIn Distributed Graph Service

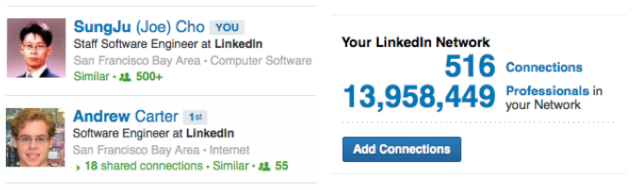

SungJu (Joe) Cho YOU
Staff Software Engineer at LinkedIn
San Francisco Bay Area • Computer Software
Similar • 500+
Andrew Carter 1st
Software Engineer at LinkedIn
San Francisco Bay Area • Internet
18 shared connections • Similar • 55
Your LinkedIn Network
516 Connections
13,958,449 Professionals in your Network
Add Connections

Fig. 1. Examples of a search result page and size of user's network in LinkedIn. The distance badge in search result supports up to 3 degrees ("You" means distance 0).

LinkedIn is the largest professional social network with more than 400 million members. The power of LinkedIn is derived from the insights that can be computed about each member's network. For example, as shown in the Fig. 1, when searching on the LinkedIn site, users are able to see how many degrees, up to 3, away other users are. Additionally LinkedIn displays the size of a user's direct connections, as well as how many professionals are in the user's network. At LinkedIn, these metrics are computed online using our distributed graph service. The distributed graph service allows other services at LinkedIn to perform graph operations such as retrieving a member's connections, computing common connections, calculating distance between members, and computing the size of a member's network. Although these operations are relatively simple graph operations, the immense scale at which these operations must be done makes them complicated.

Besides the connections among members, the member's network also contains other edge types including employments, educations, job titles, skills, LinkedIn groups, following relationships, etc. The distributed graph service handles hundreds of thousands of queries per second and scales to hundreds of millions of nodes and tens of billions of edges.

A. Architecture

The Fig. 2 shows the LinkedIn graph service architecture. GraphDB is an in-memory key-value store based graph database, which requires partitioning the entire graph data into multiple GraphDB instances. The Graph API Service is a stateless query-processing layer that handles all incoming queries to the graph service. We provide SQL similar Graph Query Language (GQL) to allow graph service users to retrieve data from the social network in a flexible manner. Both GraphDB and the Graph API service support GQL query processing. The Graph API Service receives GQL queries from graph service users, parses the GQL query, and then executes the query and returns a result set to the caller by fetching data from Graph DB. GraphDB also uses GQL query engine, which

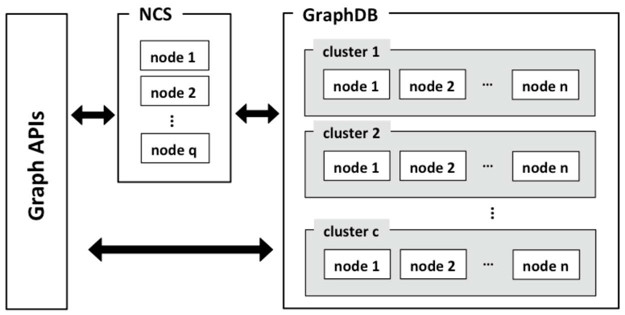


Fig. 2. The LinkedIn distributed graph service architecture consists of three major subsystems: GraphDB, NCS, and API layer. GraphDB is a horizontally partitioned key-value store for members' edges. NCS serves requests requiring a member's 2^{nd} degree network. The API layer can send simple requests directly to GraphDB if they don't require the 2^{nd} degree network or route more complicated requests to NCS.

enables efficient query processing by balancing data operations between the Graph API Service and GraphDB.

The Network Cache Service (NCS) maintains a cache of members' 2^{nd} degree network, and provides specialized graph service such as computing a member's network, distance badging, path finding, and calculating network size. The computation of a member's 2^{nd} degree network has high processing cost because the query must fan-out requests to many GraphDB partitions. For many members, the member's 1^{st} degree network size is large enough that the member's 1^{st} degree connections are scattered in all partitions. Therefore, the Graph API Service delegates the processing of these high cost queries that need 2^{nd} degree network to NCS. Due to the high frequency of these specialized service calls, NCS has a cache hit ration over 95%.

B. Data Partitioning/Replication

GraphDB uses a hash-based horizontal partitioning strategy where the key is the hash value of graph node's ID. In the partitioning, the number of partitions is much larger than the number of data store nodes and each node serves multiple partitions. This allows us to easily add more GraphDB nodes by reassigning subsets of partitions instead of repartitioning the entire dataset.

This approach needs a routing table containing the mapping information between partitions and GraphDB nodes, however the routing table has negligible memory overhead because the number of partitions is small relative to the number of nodes in the graph. To provide scalability and resiliency, we replicate a set of GraphDB nodes, which is called GraphDB cluster. Each cluster contains a complete set of all partitions. This approach allows maintenance operations without service interruptions.

For the purpose of maintenance operations such as software update or adding more GraphDB, one cluster can be safely taken out of service because the remaining clusters can handle any incoming queries.

We use ZooKeeper[9] to store the routing table containing the partition to GraphDB node mapping information. Whenever the routing table is updated, ZooKeeper broadcasts the changes to all clients including GraphAPIs, NCS and

GraphDB. In each client node, Norbert maintains the routing table by listening the update events from ZooKeeper. Norbert performs client-side load balancing by using a round-robin mechanism. The routing table has partition id as the key and the set of GraphDB node as the value. Since each cluster has only one GraphDB data store node per partition, the size of the value set is the number of clusters. The round-robin load balancer uses queues to stores the set of GraphDB nodes per partition. Per incoming query, the load balancer computes the partitions and chooses GraphDB nodes from the head of each partition queue. After finishing this process, the selected GraphDB nodes are returned to the tail of each partition queue.

C. Graph Query Types

The LinkedIn graph service supports various graph retrieval queries that we categorized into three types based on the processing requirements. The first type are single key lookup queries that need data from only a single GraphDB node Typical single key lookup queries are computing a member's 1^{st} degree network, the schools a member attended, and/or the companies the member worked at. Graph APIs can generate a single request to a single GraphDB node to fulfill these queries.

The second type are multi key lookup queries that need data from multiple GraphDB nodes. The specialized services in NCS - network sizes, path-finding, and distance badging queries - leverages a member's 2^{nd} degree network, and computing the 2^{nd} degree network requires multiple GraphDB nodes. The 2^{nd} degree network consists of two sorted set: 1^{st} hop members and 2^{nd} hop members sorted by member ID. To compute the 2^{nd} degree network we first fetch the members 1^{st} degree and then request the 1^{st} degrees of all of the 1^{st} degree connections and union the results. NCS makes fan-out requests to multiple GraphDB nodes concurrently and merges the results together into one sorted set. In LinkedIn's professional social graph, most active member's 1^{st} degree network size is large enough to require fan-out requests to almost every GraphDB nodes.

The last type is common entity queries that perform intersection operations on multiple sets. This is similar to the keyword search query done by inverted indexes. For example, to find the common members connected to two given members, an intersection operation should be performed on the members' 1^{st} degree network. One option of processing this query is that a client fetches the two members' 1^{st} degree networks and computes the intersection on the client node.

Another option is that the client distributes the query to every GraphDB data node and each performs a partial intersection using per-partition inverted indexes mapping value (1^{st} hop member) to key (member). If the size of each member's 1^{st} degree networks is sufficiently large, the latter option is better because it reduces the network transfer overhead and allows distributed computation. For this option, the larger the fan-out the more efficient the query is because each request workload and response size is smaller.

In our distributed graph services, incoming queries consists of single key, multi key look up and common entity queries with the portion of 40%, 40%, and 20% respectively.

D. Issues and Motivations

Due to the fast growth rate of LinkedIn's professional social network, scalability has been a key focus of our distributed graph service. To meet the increasing incoming QPS and data size, we have added more clusters and GraphDB nodes. This scales the single key lookup queries and common entity queries. However, as we continuously add more clusters, multi key lookup queries generate large enough fan-out to cause the multiget hole problem. Once the multiget hole problem starts to dominate, the client average query processing time increases due to queuing time.

To address this, [16] implemented a set cover algorithm based routing strategy to minimize the fan-out. This effectively reduces the amount of fan-out per query, but the greedy nature of the algorithm gives unbalanced data processing overhead to the early selected GraphDB nodes during the greedy set cover iterations. The fully overloaded nodes make the tail-latency issues even worse. Various fine-grained partitioning strategies have been considered to mitigate the multiget hole issue with hope that removing the fan-out request in the multi key query processing. However, finding well-balanced partition with a minimal edge cut is difficult. Even if we find an effective partitioning strategy, keeping the minimal edge cut status in our ever-changing Online Social Network is very expensive.

Given these limitations, we believe that the hash based horizontal partitioning is the best option. Thus, Moolle is motivated by the two important factors: capability to control fan-out requests between client and data store nodes on a per query basis in order that we can fan-out less in the process of multi key lookup queries and fan-out more in the process of common entity type queries, and the ability to us a simple partitioning strategy well suited to ever-changing data. We believe that the per query basis fan-out control capability can provide opportunities of optimizing the global efficiency of the distributed graph services by considering the global performance related statistics.

III. Design

The primary goal of Moolle's design is to enable fine-grained control for the number of fan-out requests in a single query processing. The secondary goal is to have simple and efficient data partitioning strategy that provide the same or an enhanced level of scalability and resilience compared to other systems. Aligned with these goals, we designed Moolle based on three notable concepts: replicated hash-based horizontally-partitioned data stores, cluster-aware round-robin load balancing, and per query fan-out control. Based on Norbert, a round-robin load balancer, we designed a cluster-aware load balancer that limits the number of fan-out requests. In the design, the fan-out controller only needs to calculate the optimal number of fan-out requests per query. The Fig. 3 shows these three components in the architecture of Moolle. In this section, we describe the basic idea of controlling fan-out requests first, and then we detail the design of the data partitioning strategy, cluster-aware load balancer, and fan-out controller.

A. The basic idea of fan-out control

For a 2^{nd} degree network lookup query, the number of fan-out requests depends on the number of keys to look up in the

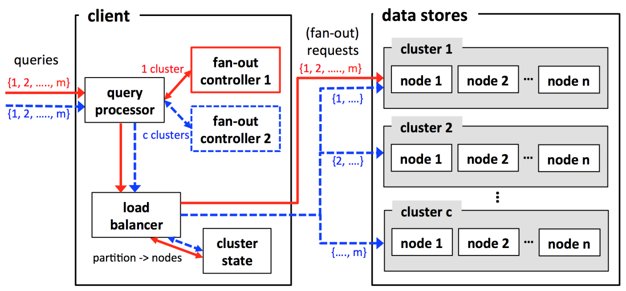


Fig. 3. An overview of the Moolle system and fan-out control examples. The fan-out controller and cluster-aware load balancer are located in the client. The client keeps track of the state of the multi-cluster partitioned data stores. The red solid line in the diagram shows an example of fan-out to only one cluster. The red solid box is the fan-out controller that selects cluster 1 and fans-out transactions to only nodes in that cluster. The blue dashed line in the diagram shows the example of the fan-out to c clusters. The blue dashed box is the fan-out controller that selects all c clusters and fans-out transactions to nodes in all c clusters.

query. For example, if the query has k keys to lookup, then the query processor in the client needs to fetch k values from data stores. In a single query, Norbert can bundle multiple lookup requests to the same GraphDB node into one request. The number of fan-out request is the number of data store node participating in the query processing. To understand the relations between the number of keys and the number of participating data store nodes, we consider the probability, P, that processing a query involves a given node as

$$P = \left(1 - \left(\frac{n_{candidate} - 1}{n_{candidate}}\right)^k\right), \tag{1}$$

where k is the number of keys per query and $n_{candidate}$ is the total number of candidate nodes that can participate in the query processing. The expected number of nodes participating is the number of candidate nodes, $n_{candidate}$, times the probability that each node will be selected:

$$E = n_{candidate}\left(1 - \left(\frac{n_{candidate} - 1}{n_{candidate}}\right)^k\right). \tag{2}$$

Note that if $n_{candidate} \ll k$ then $E \to n_{candidate}$. Without considering the 1^{st} degree network lookup query, E is the number of fan-out requests in the query processing. Most active LinkedIn member's 1^{st} degree network size is sufficiently large to make $E \to n_{candidate}$. Therefore, we realized that controlling the number of candidate nodes to participate in the query processing would limit the number of fan-out requests. Because each data store cluster has a complete set of keys, a single cluster can handle any incoming query. With a sufficiently large k, the number of fan-out requests is the number of data store nodes in the cluster. By extension, if we have $c_{candidate}$ clusters of n_{nodes} nodes involved in the query processing we can calculate the expected number of fan-out requests as

$$E = c_{candidate}n_{nodes}\left(1 - \left(\frac{c_{candidate}n_{nodes} - 1}{c_{candidate}n_{nodes}}\right)^k\right). \tag{3}$$

Note that if $c_{candidate}n_{nodes} \ll k$ then $E \to c_{candidate}n_{nodes}$. Therefore, controlling the number of candidate clusters to participate in the query processing can control the number of fan-out requests.

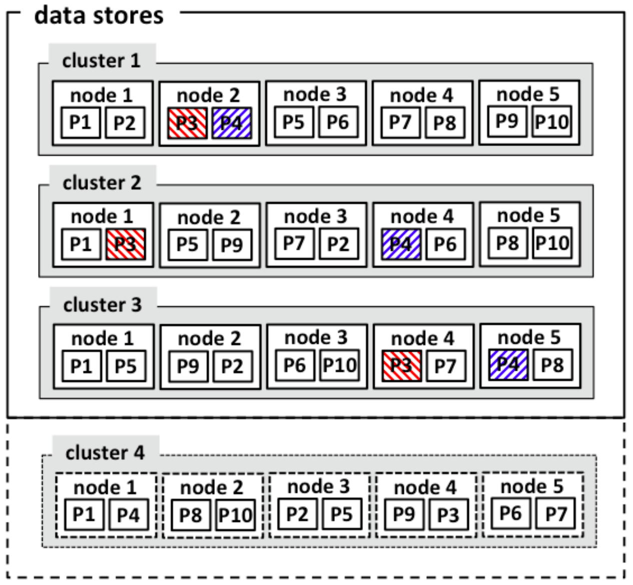


Fig. 4. An example of Moolle's partitioning consists of 3 clusters of 5 nodes with 10 partitions. The dotted box shows an example of the easy addition and deletion of clusters in this partitioning strategy.

B. *Partitioning*

In designing the partitioning for Moolle, it is important to be able to scale the size and number of the data stores in order to have easy operationality. The replicated, hash-based, horizontal partitioning used in our distributed graph system meets these requirements. The Fig. 4 shows the example of partitioning in Moolle.

The whole key space is divided into multiple partitions using a pseudo-random hash function. For the resiliency and scalability, we replicate the cluster so that losing one data store node does not interrupt the entire service and simply adding more clusters can easily scale the system. In each cluster, the partitions are assigned to nodes in such a way that the nodes in a cluster cover all the partitions and no two nodes in a cluster serve the same partition. We select the number of partitions to be much larger than the number of nodes in order to avoid repartitioning if we add nodes to a cluster. Therefore, any single cluster is enough to process any incoming query. Each node stores the key-value pairs for each key that is in its partition. To prevent hot spots and minimize the chances of cascading failures, partition assignment to nodes in different clusters is not the same. This allows the set cover based routing algorithm to effectively select minimal sets.

In our system we use separate directories per partition, and use Apache ZooKeeper [9] to keep track of the global state. We can increase the number of nodes in a cluster, in order to decrease the memory footprint per node, in an online manner. We take down a single cluster at a time, add one or more new nodes, move some partitions from the old nodes to the new nodes, and then update the state in ZooKeeper. Similarly, we can add another cluster by copying the data from an existing cluster onto the new nodes and updating the state in ZooKeeper. We can scale both in the size of the data set, and in the request rate without taking down the service. The clustering and partitioning information is available to the client so that the client can select clusters and nodes to which to send the fan-out requests. In the client, Norbert collects the information from ZooKeeper and constructs routing table where the key is the partition id and the value is the set of data store nodes having ownership of the partition id. Per partition id, the number of data store nodes is same as the number of clusters. If some data store nodes are down due to either a system failure or maintenance operation, ZooKeeper marks the nodes down and broadcast the information to all the clients and data store nodes, and these update their routing table to mark the node as unavailable.

C. Cluster aware load-balancer

Moolle's core component, the cluster-aware load balancer has the capability of limiting the number of clusters to participate in the query processing, and exposes this control as an API parameter. If the user sets the number of cluster, $c_{selected}$, the load balancer selects $c_{selected}$ clusters randomly. With the set of partition ids generated from a given set of keys to lookup, the load balancer iterates over the partition ids and selects a node from the routing table and assigns the key to the selected node. Once a node is selected and assigned, the node is returned back the routing table queue to achieve round-robin manner. If a node has multiple key assignments, the load balancer bundles them into single request to the node.

Compared to the set cover algorithm, this approach achieves a more even distribution of keys across selected nodes. If the number of input keys is sufficiently large, the set cover algorithm assigns a higher number of keys to the nodes selected earlier than those selected later, which can cause long tail request response time. However, Moolle's load balancer distributes the keys across selected nodes in a more even manner, which effectively prevents the long tail request response time.

For even better balanced key assignments, we designed an assignment adjustment where the load balancer removes key assignments from heavily assigned nodes and reassign the keys to under assigned node. We propose a node borrow concept that picks the under assigned node from the non-selected clusters.

D. Fan-out controller

We propose a user-definable fan-out controller that implements a function that determines the optimal number of fan-out requests based on various factors including the number of keys to fetch, the query type, the client side RPS, and the global RPS across data store nodes. For each query, the query processor make calls to the fan-out controller, then the controller considers the input factors and its control model to determine the optimal number of fan-out requests. For example *multi key lookup type queries* and *common entity type queries* have different processing requirements. Even queries of the same type can have different performance requirements. For example, queries in *synchronous* and *asynchronous* workflows should have different fan-out control strategies. We take advantage of this by implementing separate fan-out controllers for those various of types of queries.

Moreover, the per query basis fan-out control can optimize the global and system efficiency by considering the global and local status metrics. For example, while applying aggressive fan-out requests for one type of queries, it is possible that the aggressive fan-out would generate high RPS and cause the multiget hole problem. If we can derive the threshold of RPS that causes the multiget hole problem, we can implement a controller that controls the number fan-out requests based on the measured RPS in client side. It is also possible to apply various feedback control system practices such as a PD controller[15] to enhance the entire system stability.

The Fig. 3 shows the query processing workflow in Moolle system. Once a query arrives to the query processor, the query processor determines which fan-out controller to use to compute the amount of fan-out requests based on the query type. After the controller chooses the number clusters to use, the query processor passes the control to the cluster-aware load balancer. The load balancer would select either 1 or c clusters to participate in the query processing and send requests to the data store layer. For 1 cluster, the number of fan-out requests is limited to the n and for c clusters, limited to the cn.

IV. Fan-out Controller Design

Moolle is a generic distributed data store system that can solve the issues of fan-out requests in general distributed data services. The fan-out controller can achieve the desired fan-out by considering the types of incoming queries and global performance related metrics.

In this section, we describe the design of the fan-out controllers used in our distributed graph service in our production environment to demonstrate how Moolle can solve the issues of fan-out in distributed data services. To understand the types of fan-out requests and their processing requirements, we describe the algorithms of important queries and the production system issues. To satisfy the requirements of the fan-out requests, we describe the basic fan-out controllers. Then, to achieves the global performance optimization, we describe the advanced fan-out controllers. Finally, we evaluate each fan-out controller design in section V.

A. Query Analysis

LinkedIn's distributed graph service supports different queries including simple edge set look ups, node centric graph traversals such as distance badging or path finding, network size computation, and common entity computation. Most of the query processing uses specialized bi-directional breadth first graph traversal algorithms. To optimize the performance and efficiency of the query processing, NCS serves member connections distance badging, path finding, and network size computing queries by caching each members 2^{nd} degree network. The Graph API Service processes other queries including simple edge set lookups and common entity computation and checking by generating fan-out requests to GraphDB nodes directly. In this section, we describe only the distance badging and common entity checking queries because they generate the most important types of fan-out requests.

1) Distance badging query: The distance badging query computes the distance, up to three degrees apart, between a single source graph node, s and multiple destination nodes,

D based on the breadth-first graph traversal algorithm. The Graph API Service delegates the query processing to NCS in order to leverage the members 2^{nd} degree network cache. The members 2^{nd} degree network cache consists of 1^{st} and 2^{nd} degree connections that are sorted integer sets containing only member ids. The Algorithm1 shows the distance badging implementation in NCS.

Algorithm 1 Distance Badging

```
1:  procedure GETDISTANCE($s$, $D$)
2:      $R \leftarrow$ map($id$, $distance$)
3:      $N \leftarrow$ GETNETWORKCACHE($s$)
4:      for each $id \in D \cap N[1]$ do
5:          $R[id] \leftarrow 1$
6:          $D = D - id$
7:      for each $id \in D \cap N[2]$ do
8:          $R[id] \leftarrow 2$
9:          $D = D - id$
10:     $F \leftarrow$ FETCHCONNECTIONS($D$)
11:     for each $id \in D$ do
12:         if $F[id] \cap N[2] \neq \emptyset$ then
13:             $R[id] \leftarrow 3$
14:         else
15:             $R[id] \leftarrow$ UNKNOWN
    return $R$
16: procedure GETNETWORKCACHE($s$)
17:     $N \leftarrow$ map($degree$, $connections$)
18:     if Cache[$s$] $\neq \emptyset$ then
19:         $N \leftarrow$ Cache[$s$]
20:         if $N$ is expired then
21:             Cache[$s$] $\leftarrow$ ASYNCFETCHNETWORKS($s$)
22:     else
23:         $N \leftarrow$ FETCHNETWORKS($s$)
    return $N$
```

The GETDISTANCE function (in Algorithm 1) computes up to two degree apart only using the s member's network cache, Cache[s] using intersection of D and 1^{st} and 2^{nd} degree connections in the cache (line 3 to 9). To compute third degree distances, it must fetch the 1^{st} degree connections of each the remaining destination ids from GraphDB (line 10) and intersect the 1^{st} degree with the 2^{nd} degree connections in the cache (line 11 to 15). The network cache is a look through cache so in case of cache miss, the processing must wait until the cache is created.

The GETNETWORKCACHE (in Algorithm 1) function retrieves the cache store to see if s member's network cache exists (line 18). In the case of cache miss, it fetches s member's network cache by calling the FETCHNETWORKS function (line 23). In case of stale cache hit, it returns the stale cache, then asynchronously updates the cache by calling the FETCHNETWORKS function (line 21). The ASYNCFETCHNETWORKS function implements the same algorithm as the FETCHNETWORKS function except the caller asynchronously handle the response using a separate thread.

The FETCHCONNECTIONS function call fetches the 1^{st} degree connections from multiple GraphDB nodes if the D contains multiple ids, which is the first fan-out request type. Because the result should not be unioned together, we name this type of fan-out requests as ***non-union fan-out requests***.

Algorithm 2 Network Cache Creation

```
1: procedure FETCHNETWORKS($s$)
2:     $N \leftarrow \emptyset$
3:     $F \leftarrow$ FETCHCONNECTIONS($s$)
4:     $S \leftarrow \emptyset$
5:     for each $id$ in $F$ do
6:         $S \leftarrow S \cup$ FETCHCONNECTIONSUNION($id$)
7:     $N[0] \leftarrow s$
8:     $N[1] \leftarrow F$
9:     $N[2] \leftarrow S$
   return $N$
```

The FETCHNETWORKS function (in Algorithm 2) fetches the 1^{st} and 2^{nd} degree connections from multiple GraphDB nodes by generating fan-out requests. It consists of two steps: fetching 1^{st} degree connections (line 3) and fetching 2^{nd} degree connections, the 1^{st} degree connections of each member in the 1^{st} degree connections (line 6).

Because the network cache stores the 1^{st} and 2^{nd} degree connections in sorted integer sets, all the 1^{st} degree connections of the 1^{st} degree connections must be unioned together. Each GraphDB node returns a sorted integer set after performing a local union operation on the 1^{st} degree connections of all the ids with which the GraphDB node receives the FETCHCONNECTIONSUNION (line 6) function call. The GraphDB node local union reduces network traffic significantly because of the significant overlap between the 1^{st} degrees of the 1^{st} degree connections of the LinkedIn members. The FETCHCONNECTIONSUNION function generates fan-out requests and performs a union of the entire response, so, we name this type of fan-out request as ***union fan-out requests***.

The FETCHNETWORKS function call is handled synchronously under cache-miss (line 23 in Algorithm 2) and asynchronously under stale cache-hit (line 21 in Algorithm 2). Therefore, we break down the ***union fan-out requests*** into ***synchronous union fan-out requests*** and ***asynchronous union fan-out requests*** because each of them has different processing requirements.

2) Common entity checking: The common entity checking query has lots of applications including determining if two members have common connections, share the same school or company, or follow the same influencer or company. Processing this query leverages the inverted index in each GraphDB node, which is identical to a conventional search system's query processing using distributed inverted indices. To process this query, the Graph API Service node generates the same fan-out common entity checking requests to all the partitions, each of which evaluates the requests based on its own data. The query processing has the benefit of early termination techniques; if at least one response confirms that two members share a common entity in one data partition, the query processor can answer the query without waiting for all the responses of the fan-out requests.

The HASCOMMONENTITY function (in Algorithm 3) chooses the set of nodes N such that the union of the set of partitions in each node covers all the data partitions (line 2 to 6). Then, it dispatches fan-out requests to all the selected nodes concurrently (line 8 to 9). Then, it waits on the response

Algorithm 3 Common Entity Checking

```
1:  procedure HasCommonEntity($s$)
2:      $PN \leftarrow$ map($partition$, $N$)
3:      $N \leftarrow \emptyset$
4:      for each $partition$ in $P$ do
5:          $node \leftarrow$ next node in the $PN[partition]$
6:          $N \leftarrow N \cup node$
7:      $Q \leftarrow \emptyset$
8:      for each $node$ in $N$ do
9:          ConcurrentHasCommonEntity($Q$, $node$)
10:     $total \leftarrow N.size$
11:     while $total > 0$ AND $Q \neq \emptyset$ do
12:         $r \leftarrow$ head of $Q$
13:         if $r ==$ true then return true
        return false
```

queue until all the responses return (line 11 to 13). In case that a response confirms the connectivity, it can terminate the processing without waiting for the other responses (line 13). Note that as the common entity checking supports member-to-company/school/followers type connections, it doesn't leverage NCS because NCS only stores the member-to-member type connections in its cache.

The ConcurrentHasCommonEntity call is another type of fan-out requests, which we refer to as ***early terminable fan-out requests***.

B. Production system analysis

During the design of fan-out controllers for different types of fan-out requests, we closely investigate the production distributed graph system. In this section, we discuss important aspects of the system that direct the fan-out controller design.

The analysis in this section uses production distributed graph service system. LinkedIn has multiple collocated data centers, each of which serves web services with the full stack of web services and applications. Each data center contains an independent distributed graph service that has different service configurations. The distributed graph service in this analysis uses 200 GraphDB nodes with 10 clusters and 20 nodes in each cluster. Note that each data center may have a different number of clusters and a different number of nodes in each cluster. Based on the scalability plan, the number of GraphDB nodes and clusters can vary overtime.

1) Tail latency: In distributed data services, tail latency is a well-known issue that also appears in our distributed graph service. This section describes the pattern of the tail latency in production environments.

The Fig. 5 shows the latency distributions of single key lookup request between a NCS and the GraphDB nodes. The single key lookup request fetches a single members 1^{st} degree connections from one GraphDB node. Even for members with the maximum number of connections, the median latency of a single key lookup query is under 2 millisecond, which means that the size of response in this evaluation is small enough not to give much impact on the latency. However, the 99th percentile is 21 milliseconds and the maximum is 323 ms. This pattern is the long tail latency.

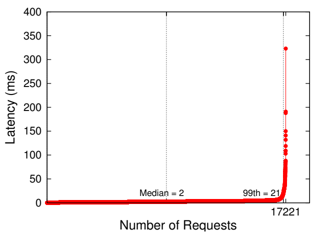


Fig. 5. Long tail latency pattern between NCS and GraphDB.

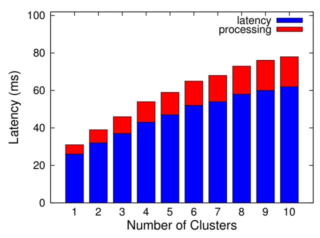


Fig. 6. Non-union fan-out requests latency between NCS and GraphDB.

Mitigating the heavy tailed latency pattern issue is one of most important tasks what we have, however, in this paper, we focus on how the tail latency influences the processing requirements of each type of fan-out requests.

2) Fan-out request query latency: Considering only a single query that generates fan-out requests, the latency of the query processing consists of the maximum latency on the server side among the fan-out requests plus the processing time of the outgoing requests and incoming responses on the client side.

$$L = L_{client}\left(n_{selected}\right) + \max_i \left(L_{server_i}\left(\frac{k}{n_{selected}}\right)\right) \tag{4}$$

In this equation $n_{selected}$ is the number of nodes selected in the query processing and k is the number of keys per query. L_{client} is a latency function of the client side work, and L_{server_i} is the latency function of an individual server. The right term represents the latency of each request, which consists of network transfer latency plus server side request processing latency.

An aggressive fan-out increases the $n_{selected}$; as a result, the left term increases and the right term decreases. Intuitively, an aggressive fan-out requests should achieves better latency because the right term is larger than the left term in general.

However, due to the tail-latency pattern, an aggressive fan-out requests has a higher probability of having a request with long latency. Therefore, large, aggressive fan-out requests should have worse latency than small fan-out requests. We verify this by evaluating the non-union fan-out requests between a NCS node and GraphDB nodes, as shown in Fig. 6.

Fig. 6 shows that the average latency of non-union fan-out requests increases as more aggressive fan-out is used. For the data in Fig. 6, we measure the latency of the fan-out requests between a NCS node and GraphDB nodes by generating the same set of distance badging queries to NCS and varying the number clusters to fan-out using the fan-out controller. The blue portion of each bar indicates the right term and red portion indicates the left term of the equation 4.

C. Basic fan-out controllers

In section IV-A, we describe the important types of fan-out requests used by the LinkedIn distributed graph service. Each type of fan-out requests requires a unique fan-out control strategy because each has different processing requirements. Our production system uses several fan-out controllers per query type based on Moolle. This section describes the design of the fan-out controllers. We believe that the fan-out controllers are general enough to apply to other general distributed data services.

1) Non-union fan-out controller: In general, the non-union fan-out request is latency sensitive. For example, distance badging query processing must wait for the response of all the non-union fan-out requests (in Algorithm 1). Considering that the equation 4 is the latency model of this request type, the fan-out controller for this type of query should minimize the degree of fan-out requests to optimize the latency. Note that this fan-out strategy also reduces the client side overhead, the left term in equation 4.

Therefore, we design the non-union fan-out controller in order to minimize the degree of fan-out requests by selecting only one cluster to fan-out to. This fan-out controller generates at most 20 fan-out requests in our evaluation production system because each GraphDB cluster has 20 GraphDB nodes.

2) Synchronous union fan-out controller: Synchronous union fan-out requests are very similar to the non-union fan-out request except that the union fan-out request returns one single set after applying union on entire 1^{st} degree connections. In the case that a GraphDB node receives a request with multiple member ids due to request bundling, the node returns a single set after unioning all of the 1^{st} degree connections. Therefore, this type of query can generate additional overhead for the response processing on both the client side and server side.

The latency model of the union fan-out requests is very similar to equation 4. In both the client and server side, performing additional union operations would increase the processing time, however the heavy tailed latency pattern hides the impact of the processing time of the union operations. Therefore, for this type of query, minimizing the degree of fan-out requests should achieve best latency.

3) Asynchronous union fan-out controller: A query that generates asynchronous fan-out requests do not need to wait for the responses. For example, during a distance badging query processing, if NCS updates the network cache asynchronously, the query does not need to wait for the response of the asynchronous network cache update. Therefore, the asynchronous fan-out requests is not latency sensitive.

Instead, for the asynchronous union fan-out requests, we develop a new fan-out controller that balances the computation overhead on both the client and server. The union operation causes computation overhead in both client side and server side, which means that minimizing the degree of fan-out requests pushes more computation overhead to the server side.

$$P = P_{client}\left(n_{selected}\right) + \max_i \left(P_{server_i}\left(\frac{k}{n_{selected}}\right)\right), \quad (5)$$

where P_{client} and P_{server} are the computation penalty function in client and server side. In our distributed graph service, we determine the client side and server side penalty function based on empirical testing. The penalty function $P_{client}(x)$ and $P_{server}(x)$ are proportional to $xlog(x)$. This is the running time of our union algorithm.

The asynchronous fan-out controller controls the $n_{selected}$ to minimize the P. This fan-out controller dynamically controls the degree of fan-out based on the number of keys, k. For example, in our production system, queries with less than 450 keys are sent to one cluster, those with about 1,800 keys fan-out to five clusters, and those with more than 3,000 keys fan-out to all 10 clusters.

The set cover based approach in [16] can cause GraphDB side intermittent latency increases because the greedy fan-out pushes heavy computation overhead to the GraphDB nodes. However, the dynamic fan-out control does not cause latency increases because it balances the computation overhead between the client and server.

4) Early terminable fan-out requests: A query that generates early terminable fan-out requests can finish as long as at least one of the requests satisfies the query. Therefore, in the case that the query can terminate early, aggressive fan-out requests reduces the latency because each GraphDB nodes need to process less number of data partitions. However, if the query must wait for all the requests, aggressive fan-out will increase the latency due to the heavy tail latency pattern.

Therefore, we can choose a fan-out controller based on the expected percentage of queries that will terminate early. For example, for a query type that has higher probability of early termination, aggressive fan-out controller can optimize the overall latency, while for a query with lower probability, minimizing fan-out can optimize the latency.

D. Advanced fan-out controllers

In addition to the type of query and the processing requirements, the fan-out controller can use statistical information that predicts the stability of components in distributed data service. In this section, we consider the early terminable fan-out requests with a high probability of early termination and demonstrate two advanced fan-out controllers that consider the client side QPS and global RPS.

1) Client QPS based fan-out controller: Although aggressive fan-out can help reduce latency for an individual query, if the client is under heavy load, this strategy may produce negative consequences such as the multiget hole problem. Each client has a maximum throughput at which it no longer process requests/responses and instead either put the requests/responses in a queue or drops them. If the client has to enqueue them, then the latency of the query becomes the sum of the latency of the processing time and the latency of the queue wait time. Due to the additional queue wait time, the entire system's QPS throughput decreases. To handle this case, a proper strategy needs to consider the aggregate load on the client side and the impact of additional fan-out to the request rate per client.

For an incoming QPS, we can easily estimate the expected RPS.

$$E_{RPS}\left(t\right) = cnC_{QPS}\left(t\right) \quad (6)$$

$E_{RPS}(t)$ is the expected requests per second and $C_{QPS}(t)$ is current incoming query rate. c is the number of clusters and n is the number of nodes in each cluster. If we know that a particular client performs well at a desired request per second,

D_{RPS}, the best fan-out control is the most aggressive fan-out that doesn't surpass D_{RPS}. As a result, we can achieve the best fan-out in terms of the number of clusters, $c_{selected}$ as follows:

$$c_{selected} = min\left(c, \frac{cD_{RPS}}{E_{RPS(t)}}\right) \tag{7}$$

$c_{selected}$ is the number of clusters that provides the lowest latency without surpassing the desired requests rate. In general, we consider the D_{RPS} as the maximum request processing throughput of a client in terms of the number of requests per second. From LinkedIn distributed graph service, we determine D_{RPS} by throughput performance testing.

2) Global Requests Per Second: Similar to the client side processing limitations, there are also server side processing limitations. If a server receives a higher request rate than its throughput, the request processing time suddenly increases due to queue time and processing interference. We believe that the fan-out control can prevent this situation by reducing the number of fan-out requests in response to the higher QPS.

From equation 6, we can generalize the sum of request per second in entire data store layer.

$$\sum E_{SRPS}(t) = cnmC_{QPS}(t), \tag{8}$$

where m is the total number of client node and E_{SRPS} is the expected requests per second in one server node. If cn is the total number of server nodes, we can compute the E_{SRPS} as follows:

$$E_{SRPS}(t) = mC_{QPS}(t) \tag{9}$$

Using this equation we can expand the equation 7

$$c_{selected} = min\left(c, \frac{D_{RPS}}{nC_{QPS}(t)}, \frac{cD_{SRPS}}{mC_{QPS}(t)}\right), \tag{10}$$

where D_{SRPS} is the desired requests per second in one single server, which can be measured from performance test similar to the D_{RPS}. From equation 10, we can implement a controller that accounts for both client and server side throughput limitations by choosing the optimal number of clusters for fan-out requests.

V. Evaluation

In LinkedIn's distributed graph service, the use of a naive round-robin strategy caused scalability problems because the fan-out scaled linearly with the number of replicas. When the number of requests exceeds the maximum aggregate throughput of the distributed graph data store, the overall latency of the service increases because of the request queuing time in either the client or server side.

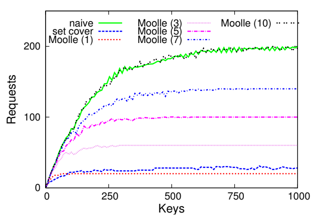
To resolve this problem, we applied a greedy set cover that reduced the number of requests. However, this can generate unbalanced requests to the data store when the system has to handle a single request that requires fetching a large number of entries. Using Moolle, we resolved both of these issues by optimizing the fan-out control based on our production environment.

In this section, we demonstrate the Moolle's fan-out controllability and production system performance improvements by comparing to the naive round-robin strategy and the greedy set cover algorithm. For the evaluation, we use production LinkedIn graph service system with 10 clusters of 20 nodes per cluster running GraphDB.

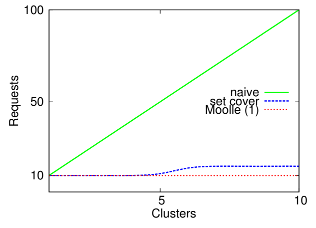
A. Fan-out control with Moolle

In this section, we demonstrate the effectiveness of the fan-out control that is the primary contribution of Moolle. We also demonstrate how Moolle can help to avoid the multiget hole problem, preventing CPU bounding based on the number of requests. In this evaluation, we generated fan-out requests that have no server side processing. We measured the impact on the number of requests as a relationship between both the number of keys in a query and the total number of clusters.

Besides both minimizing and maximizing fan-out at the same level as the set cover and naive strategies respectively, Moolle provides finer grained control of the fan-out based on the number of clusters (Fig. 7(a)). For evaluations, we implement a static fan-out controller that limits the number of fan-out requests by limiting the number of clusters. For example, for the Moolle (3), the static fan-out controller limits the fan-out requests to only 3 clusters. The Moolle (1) and (10) minimize and maximize fan-out respectively. Moolle (3) , (5) and (7) shows the finer fan-out controls.



(a) Controllability



(b) Number of Requests

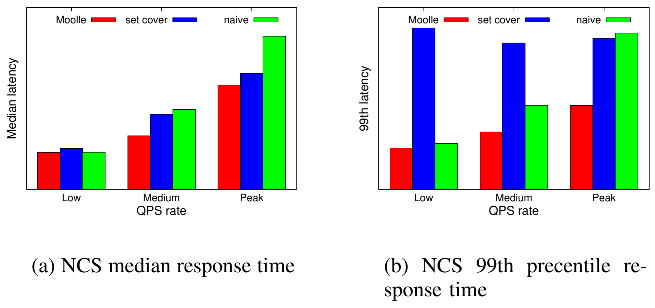
Fig. 7. 7(a) shows the changes of the number of fan-out requests as the number of keys increases in a single query that generates fan-out request. 7(b) shows the changes of the number of fan-out requests as the number of data stores increases.

To mitigate the multiget hole problem, we need to minimize the number of fan-out requests. We measure the overhead using the number of requests per query, which is the same metric as in [14]. The Fig. 7(b) shows that both Moolle and set cover strategies mitigate the multiget hole issue because they do not increase the number of requests as the number of data stores increases. Note that after 5 clusters, Moolle can even strictly limit the number of requests compared to the set cover strategy.

B. Basic fan-out controllers in NCS

For production NCS, based on the design in IV-C and IV-D, we implement fan-out controllers for the three types of fan-out requests. For the non-union and synchronous fan-out requests type, the fan-out controllers select one cluster to minimize the number of requests. For the asynchronous fan-out requests, the fan-out controller selects the number of clusters in order to minimize the penalty P in equation 5.

For the evaluation purpose, we set up one NCS node to communicate with entire production GraphDB nodes that handle real production traffic. The client generates production traffic to the NCS node by replaying a production NCS request log.



(a) NCS median response time (b) NCS 99th precentile response time

Fig. 8. 8(a) and 8(b) shows the changes of the median and 99th percentile response time of NCS with different fan-out control strategy at different QPS level.

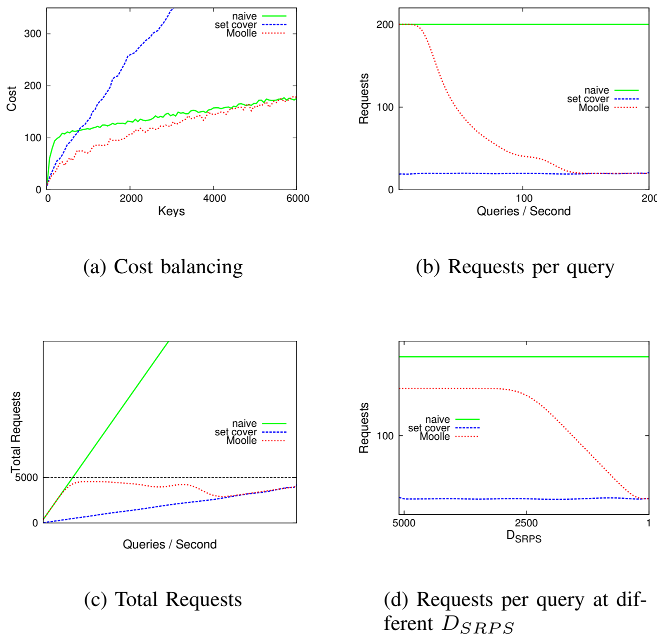
1) Latency: Moolle enhances the throughput of NCS through better performance than either the set cover or naive strategy in both median and 99th percentile response time. Compared to the naive strategy, it improves median response time by effectively optimizes the fan-out requests. Compared to the set cover approach, it improves 99th percentile response time through better key distribution among fan-out requests.

As the Fig. 8(a) shows, at medium and peak traffic, Moolle's response time is about 35% less than naive strategy. The set cover strategy improves median latency by approximately 5% at medium traffic and 25% at peak traffic compared to the naive strategy. However, the set cover's the uneven distribution of keys among fan-out requests can cause high latency by increasing the right term in the equation 4, which increases 99th percentile latency as shown in the Fig. 8(b).

This evaluation shows that Moolle effectively optimize the latency of NCS by controlling the fan-out requests based on the query.

2) Computation balancing: For union fan-out requests, a sufficiently large number of keys pushes heavy computational overhead to GraphDB nodes. Because our union operation in GraphDB requires an additional memory copy, a larger number of keys in a request to GraphDB can cause a large memory allocation, which can cause process stall due to garbage collections. That's the main motivations of the computation balancing fan-out controllers.

The asynchronous union fan-out controller can effectively balance the computational overheads between NCS and GraphDB nodes. The Fig. 9(a) shows that Moolle always generate small P in the equation 5, which means that it achieves better computation balance than set cover and naive approach. However, the set cover shows poor balancing that pushes computational overheads to the GraphDB nodes. Once a GraphDB node is processing heavy requests, it increases



(a) Cost balancing (b) Requests per query

(c) Total Requests (d) Requests per query at different D_{SRPS}

Fig. 9. 9(a) shows the computed overhead P in equation 5 and requests per query changes as the number of vertices in a query increases. As the incoming QPS increases, 9(b) shows the changes of RPS and 9(c) shows the changes of the total number requests. 9(d) shows that the number of requests changes as the desired server side RPS changes.

queue wait time of other requests. We consider this is the source of high 99th percentile latency of the set cover strategy.

C. Advanced fan-out controllers of early terminable query

For the early terminable type of query, we want to use more aggressive fan-out control to achieve better latency without surpassing a given desired RPS in the client and server nodes based on the incoming client query rate (described in the equation 10).

For evaluation, we extend Moolle's fan-out controller using equation 6 and 10 with the D_{RPS} as 5,000 RPS in Graph API node and D_{SRPS} as 10,000 RPS in GraphDB node. The desired RPSs are determined via empirical testing. Because GraphDB nodes in the production system serve traffics from both NCS and Graph API, evaluating the early terminable type of query in production is difficult. Therefore, we develop a simulator that emulates the 10 clusters of 20 GraphDB nodes.

In addition to the advanced fan-out controllers that considers the client and server side RPS, a simple PD controller using 1.0 as proportional gain and 0.5 as derivative gain is implemented for stable control. We measure the number of requests and maximum computational overhead in GraphDB nodes per request with increasing the request rate.

Fig. 9(b) shows the change in total number of requests between client and server data stores. At less than 25 QPS, Moolle maximizes fan-out in order to reduce the latency of the early terminable queries. Around 125 QPS, Moolle minimize fan-out per request at the same level as the set cover strategy. Between 25 and 125 QPS, Moolle balances aggressive fan-out and maintaining the desired RPS. Fig. 9(c) shows that Moolle can effectively keeps the RPS under the D_{RPS}.

During the peak traffic time, GraphDB nodes receive a blend of types of queries. Under this situation, Moolle should reduce the fan-out requests for early terminable queries to improve the server side performance. At constant incoming QPS (30 queries/second), Fig. 9(d) shows that as the D_{SRPS} decreases, Moolle tries to limit the fan-out requests. When $D_{RPS} > 2,500$, client side RPS control limits the fan-out to 160 requests. However, for lower D_{SRPS}, Moolle limits the fan-out in order to not surpass the target server side RPS. This demonstrates Moolle's ability to optimize the global status of the distributed data store by considering various statistical information.

VI. Related Work

There are other approaches as well for solving the multiget hole problem and routing requests in distributed data stores. We present previous approaches with a comparison to our work.

Dynamic Partitioning: Although many online social network including Facebook and LinkedIn, use distributed hashing to balance data between servers [10], [16], there has been significant interest in partitioning the data in an intelligent fashion to preserve data locality. In [13], the graph is partitioned according to social structure using replication to ensure that all neighbors of a node are co-located on the same server as the master replica of a user. This approach suffers from a large amount of additional memory overhead for replication and increased complexity involved in repartitioning masters and determining which servers to route requests. In [3], the data is partitioned using a time component, which is not considered in our work, although it suffers from similar problems in computation and complexity in determining routing as the other partitioning approaches. Our approach would work with temporal data as we are agnostic to the data retrieved.

Set cover algorithms: The second approach is to use set cover algorithms to reduce the number of fan-out requests by selecting nodes from a set of replicas to route requests. In [16], a modified version of the greedy approximation algorithm is used to provide a logarithmic bound on the number of keys and empirically reduced both the number of requests and latency. In [14], an efficient set cover algorithm is combined with replication and retry in order to decrease the number of fan-out requests. Replicate and bundle focuses on solving the problem where the focus is purely on memory fetches with low cost per request. In cases, where processing is done on the data store node side, this can decrease performance as the work might not be balanced. Set cover is NP-complete [7], any approach must use some sort of approximation algorithm which are typically greedy and therefore highly likely to produce uneven distribution of keys in fan-out requests. Though the overall latency is correlated to the size of the request, it is mainly determined by the slowest request amongst all the fan-out requests. This result in the set cover approaches leading to worse system performance when considering a global perspective.

VII. Conclusion

In this paper, we described Moolle, a system that provides fan-out control in distributed data stores at low cost of scalability and maintainability. Moolle has been implemented in the LinkedIn distributed graph service, which serves hundreds of thousands social graph traversal queries per second. With the flexible user definable fan-out controller, we can scale different types of graph queries for a power law graph[1], by implementing several custom fan-out controllers. We are able to reduce the total the number of fan-out requests generated from our graph queries by 50% and reduce the latency per request by 30%. We conclude that Moolle increases throughput and scalability of distributed data stores with the capability of optimizing processing on a per query basis.

References

[1] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman. Search in power-law networks. *Physical review E*, 64(4):046135, 2001.

[2] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[3] B. Carrasco, Y. Lu, and J. M. da Trindade. Partitioning social networks for time-dependent queries. In *Proceedings of the 4th Workshop on Social Network Systems*, page 2. ACM, 2011.

[4] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[5] J. Dean. Achieving rapid response times in large online services. In *Berkeley AMPLab Cloud Seminar*, 2012.

[6] B. Fitzpatrick. Memcached: a distributed memory object caching system. http://http://memcached.org/, 2011. [Online; accessed June-2014].

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co Ltd, 1979.

[8] T. Hoff. Facebook's Memcached Multiget Hole: More Machines != More Capacity. http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html, 2009. [Online; accessed June-2014].

[9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[10] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[11] LinkedIn. Norbert. https://github.com/linkedin/norbert/, 2009. [Online; accessed June-2014].

[12] M. Newman. Community detection and graph partitioning. *EPL (Europhysics Letters)*, 103(2):28003, 2013.

[13] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine (s) that could: scaling online social networks. *ACM SIGCOMM Computer Communication Review*, 41(4):375–386, 2011.

[14] S. Raindel and Y. Birk. Replicate and bundle (rnb) – a mechanism for relieving bottlenecks in data centers. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 601–610, May 2013.

[15] D. E. Rivera, M. Morari, and S. Skogestad. Internal model control: Pid controller design. *Industrial & engineering chemistry process design and development*, 25(1):252–265, 1986.

[16] R. Wang, C. Conrad, and S. Shah. Using set cover to optimize a large-scale low latency distributed graph. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2013. USENIX.