Distributed Knowledge Graph Querying on Edge and Cloud

Shriram R.

Department of Computational and Data Sciences
Indian Institute of Science, Bangalore 560012 INDIA
Email: shriramr@iisc.ac.in

***Abstract*—In this mid-term period, a basic distributed knowledge graph querying system has been implemented by adopting an existing graph database engine with functionalities to fetch and store local knowledge graph, partition queries for local and remote graph database, combining results from local and remote for vertex search, edge search and shortest path search queries. Experiments were performed using a small dataset to study the performance of different types of queries and an analysis of result has been performed.**

I. System Design and Implementation

The following sections cover the system design and implementation completed so far with respect to the proposed targets for midterm,

A. Remote (Cloud) Layer

This layer consists of an in-memory graph database engine *Tinkergraph* running on the head node of *Rigel* Cluster. Integration of GoDB was explored but due to technical issues, *Tinkergraph* was chosen as the desired engine. It offers a good set of APIs to run a variety of queries related to graphs and supports different programming languages. A distributed version spanning multiple nodes will be explored in future.

B. Edge Layer

The edge layer consists of different modules each with a specific functionality. The modules are explained in detail in the section below,

1) Edge Graph Processing: The same *tinkergraph* engine used in the remote layer is spawned on the edge as a single node in-memory store. The performance of this is evaluated through experiments. A custom engine is required only if the edge based *tinkergraph* engine is found to be a bottleneck.

2) Knowledge Graph Partitioning: The logic for partitioning is based on the association of an edge with a single entity in the Knowledge graph. E.g. <India>. Given a entity, this module queries the remote server to fetch the subgraph centered at the given entity and spanning for a specified number of hops (E.g. 2). This subgraph is then inserted into the *Tinkergraph* server running locally and forms the local knowledge graph.

3) Query Partitioning: The query partitioning module divides the input query into local and remote queries and fires them against the respective servers. The input queries are provided in JSON format with custom fields for each query type. The logic is different for each type of query. It is detailed below,

- *Vertex Search* - The given query consists of three fields one for label based search, one for set of vertices with incoming edge from a given vertex and one for set of vertices with outgoing edge to a given vertex. The query is unmodified and used for local and remote since this search operation is embarrassingly parallel
- *Edge Search* - The given query is unmodified as in the previous case since it also embarrassingly parallel and consists of three fields one for label based search, one for all edges outgoing from a vertex and other for all edges incoming to a vertex
- *Path Search* - The path search queries consists of a source vertex and target vertex and finds the shortest path between them in an undirected knowledge graph. It can have three result types: Full path is in edge layer, full path is in remote layer and path crosses edge and remote layer. The logic follows these steps,
  - A local query is fired first to search for path completely contained in the edge layer
  - If the previous step returns no path, then a series of queries are fired to the remote layer with source vertex changed to *cut vertices* and target vertex unchanged. This will return a portion of shortest path in the remote server
  - A series of local queries are fired to determine the paths from original source to *cut vertices*.

4) Combining Query Results: Combining the query results is implemented as follows,

- *Vertex & Edge Search* - The result set from local and remote server search are combined using *set union* operation
- *Path search* - If the path is entirely contained inside edge, the local server result is directly used. For cross paths, The result from local server and remote server are joined at the *cut vertex* and then the shortest path is determined and provided as output

II. Experiments

The experimental setup is as follows: Microsoft Azure cloud platform will be used to run the entire edge-cloud setup. 5 D4 VMs will be used to run the cloud layer. These VMs will host the GoDB head and worker nodes. GoDB uses GoFFish v2.6 and Java v1.7. VIoLET [3] will be used to virtually emulate the entire edge layer. 20 edges of type Pi3B+ will be created and interconnected with each other. The latency and bandwidth of edge-edge and edge-cloud will be set to 1ms-50mbps and 5ms-20mbps respectively. All the devices will be running on CentOS v7.

The YAGO dataset will be the primary knowledge graph data that will be used for the experiments. Since the size of the full dataset is huge, only a fraction of the dataset will be potentially used based on the availability of sufficient cloud resources. NELL [4] could be used as a secondary dataset. The baseline system for comparison will be GoDB in cloud layer with all the edges running all the queries directly on the cloud without caching. The graph in GoDB will be partitioned using METIS with one subgraph per core.

The query workload for the experiments will be of the following types - Vertex search, Edge search, Reachability, Neighbourhood, Shortest Path and as a stretch goal, Subgraph Isomorphism.

The size of the knowledge graph that be accommodated inside the edge device will be determined through the following experiment - 10 queries of each type will be triggered on the edge for different sizes of local graph which is doubled in each iteration. The same set of query will also be triggered in cloud and the time taken in both the cases will be measured. This size should be the maximum size in which the local query time is less than the global query time for the same subgraph. This size will be used in all the below experiments as well.

Microbenchmarks

1) Query partitioning logic will incur some time to partition the query and generate a query plan. It is useful to measure this time for each query type. An experiment is performed with 100 different queries for each type being run on the edge and the time to partition is measured and plotted as a violin plot. The violin should ideally not have long tails.
2) Graph indexing time and memory has to be measured for different query types as a function of graph size. In this experiment, for each graph size, index construction time and index size is measured for each index type. The time taken and size should increase as per the time and space complexity of the index.

The experiments will test the impact of query partitioning, query caching and graph indexing. The key metric to be measured is the query latency (time to get the results).

1) **Query Partitioning:** 100 random queries of each type will be triggered in total with the queries uniformly and randomly distributed across all the edges. The time taken to complete the query in the local knowledge graph, the global knowledge graph and query completion time is measured. When compared with baseline, the query completion time should be less in the new system and the local querying time should be far less than the global time since the size of local graph will be smaller compared to the global graph. This showcases the effectiveness of Query Partitioning on the performance.
2) **Query Caching:** There will be two baselines for comparison. One is the cloud only baseline and the other is the edge-cloud setup with cache disabled. In this experiment, each edge will have 20 different queries. These queries will be executed 5 times each in random order. This is to enable repetition of the same query from a given edge device. The time taken to complete the query in local and global graph and the total time will be measured. The cache hit/miss count will also be logged for each query execution. The cache enabled version should ideally outperform the other two baselines in terms of global execution time since the global query result will be cached in the edge.
3) **Graph Indexing:** The baseline for graph indexing will be edge-cloud setup with indexing disabled on the edge. This experiment consists of 100 random queries of each type similar to the Query Partitioning experiment. The time taken to complete the local query with index and without index is measured. The outcome should be that the time taken with index should be less than the without index baseline. The effectiveness of each index for each query type will be different since problems which are NP complete like subgraph isomorphism will offer more benefit in having the index than other simple problems like vertex or edge search.

References

[1] "Titan." [Online]. Available: http://thinkaurelius.github.io/titan/

[2] "Dgraph." [Online]. Available: https://dgraph.io/

[3] S. Badiger, S. Baheti, and Y. Simmhan, "Violet: A large-scale virtual environment for internet of things," *CoRR*, vol. abs/1806.06032, 2018. [Online]. Available: http://arxiv.org/abs/1806.06032

[4] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, B. Yang, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, J. Krishnamurthy, N. Lao, K. Mazaitis, T. Mohamed, N. Nakashole, E. Platanios, A. Ritter, M. Samadi, B. Settles, R. Wang, D. Wijaya, A. Gupta, X. Chen, A. Saparov, M. Greaves, and J. Welling, "Never-ending learning," *Commun. ACM*, vol. 61, no. 5, pp. 103–115, Apr. 2018. [Online]. Available: http://doi.acm.org/10.1145/3191513