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*—Knowledge graphs with millions of vertices and edges are currently being used to enable smart devices and knowledge discovery. IoT (Internet of Things) devices are becoming the consumer and producer of such large graphs. Existing graph querying and processing systems are designed to run on commodity hardware. However, IoT applications need lightweight graph processing systems that can scale across large number of devices and handle concurrent queries with low latency. In this project, a lightweight distributed graph querying engine is proposed which can natively scale across edge and cloud devices. It supports declarative query model for vertex, edge, reachability, shortest path and subgraph pattern mining queries. The engine has a novel caching, indexing and query partitioning mechanism which enables to scale efficiently with graph size and query workload. The engine is evaluated on a large scale virtual IoT testbed with massive real-world knowledge graphs. The results indicate that the engine can scale to many devices and provide x% lower latency on a diverse query workload compared to the baseline system.**

I. Problem Description

The problem that is being solved in this project is the scaling of graph querying on edge and cloud devices which can provide low-latency query response for many concurrent queries on massive knowledge graphs. *Figure 1* shows a representation of typical edge-cloud layout. There is a hierarchical structure where edges can communicate with other edges and can directly communicate with only one cloud device. Each edge may be associated with a single fog device depending on its spatial proximity or any other parameter. The bandwidth and latency of edge-edge and edge-cloud network links are significantly different from each other.

The edge and cloud devices are heterogenous and have different compute and storage capabilities. In addition, the device failure rate varies across the layers with edge having the highest failure rate. In this project however, all the devices are considered to be 100% reliable.

Existing graph processing/querying systems are designed for commodity hardware and cannot be efficiently deployed on low powered edge devices like smartphones, smart home appliances etc. These systems do not take into account the network hierarchy and device heterogeneity

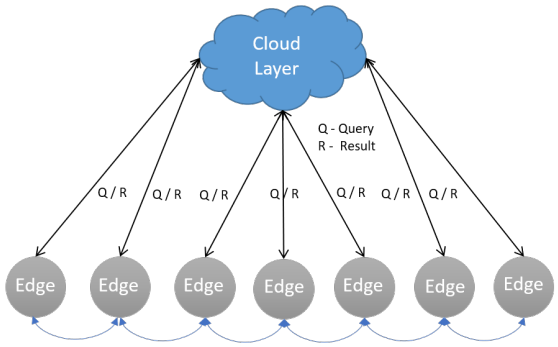


Fig. 1: Edge-Cloud Layout

that exists on the current edge-cloud layout. The proposed graph querying engine uses GoDB which is a distributed graph database as the backend and implements the novel features on top of the database.

In this project, following are the key contributions:

- **Graph Partitioning:** A novel graph partitioning approach is used which partitions the graph based on the edge device. This means that the portion of knowledge graph which is frequently required by a query source is identified and moved closer to the edge device which performs the query automatically
- **Query Partitioning:** A query planner which can efficiently partition the query and its predicates according to graph locality and placement thereby optimizing query cost is implemented
- **Query Caching:** A novel query caching system is devised which caches recent queries by edges and their corresponding results in the edge layer. It exploits the temporal and spatial locality of queries to provide low latency response. It also has a novel cache replacement policy and can quickly match an incoming query with cache.
- **Graph Indexing** Index data structures from existing literature that can accelerate the processing of queries are built in a distributed fashion and are used for pruning the knowledge graph for processing. Different type of indexes are built for each query type and are maintained dynamically as per updates to the knowledge graph

II. Novelty/Scalability

This problem is novel and has the following scalability challenges: The size of real-world knowledge graph data is massive with respect to the storage available in edges. For example, YAGO [1] which is a semantic knowledge base has about 10 million nodes and 120 million edges along with various properties. It has a total size of 170 GB (uncompressed).

Many of the graph mining problems and pattern matching (subgraph isomorphism) are NP complete which means that they cannot be feasibly computed in real time using just the edge devices.

The knowledge graphs are relatively dynamic in nature with multiple updates throughout the graph happening in a short time interval. This poses a challenge in terms of building and updating indexes rapidly.

Graph partitioning is non-trivial due to the structure and heterogeneity of edge-cloud layout. Many of the IoT applications require real-time answers whereas the existing distributed graph engines typically address batch processing queries with high latency.

An edge-cloud system deployed on a city-wide network can have on the order of thousands of devices. The query engine has to efficiently scale to this size and simultaneously provide acceptable level of performance.

Existing query engines are built on platforms or frameworks that are resource heavy. In order to run the proposed engine on the edge device, a significant optimization of the system in terms of resource requirements is necessary.

III. Related Work

Pregel [2], *Giraph* [3] and *GraphX* [4] are designed as batch mode graph processing frameworks for commodity clusters which are not suitable for interactive queries.

Trinity [5] is a distributed graph engine built as a in-memory key-value store. Though it is possible to perform interactive graph querying, the engine depends on high-speed network and requires large memory both of which are not feasible in edge-cloud.

GC [6] focusses on caching graph queries and results for graph isomorphism problems. It also provides novel cache replacement strategies for different workloads. However, it does not address other query types and is not designed for hierarchical device layout like in our case.

C-Tree [7] and *Views* [8] provide efficient indexing techniques for subgraph isomorphism queries. However, these indexes are not designed for distributed systems where the index has to be partitioned along with the graph data.

FERRARI [9] describes an efficient index for reachability and path queries. However, it is not straightforward to translate the design for our distributed setup.

GraphS [10] provides an efficient indexing technique to perform cycle detection in large distributed dynamic graphs with low latency. Our engine can adopt this technique for heterogenous device setup.

GoDB [11] is one of the closest work to the proposed engine. It is a distributed graph database which supports declarative queries on large property graphs. *GoDB* is built on *GoFFish* [12] subgraph-centric batch processing platform thereby using its scalability.

GoDB supports vertex, edge, path and reachability queries. It also come with a novel cost model which uses execution heuristics to come with a optimum query plan that minimizes latency.

However, it is not designed to work on heterogenous device layout and is not capable of running on the edge devices. It comes with indexing mechanism for filtering vertices and edges based on properties but lack query type specific indexes which can dramatically reduce the query processing time.

GoDB also do not have any caching mechanism and therefore cannot leverage on temporal and spatial locality of input queries.

Quegel [13] system provides a vertex centric programming model for processing graph on commodity clusters. It treats queries as first class citizens and uses a novel superstep-sharing execution model along with *Hub* indexing technique to improve utilization of resources and performance.

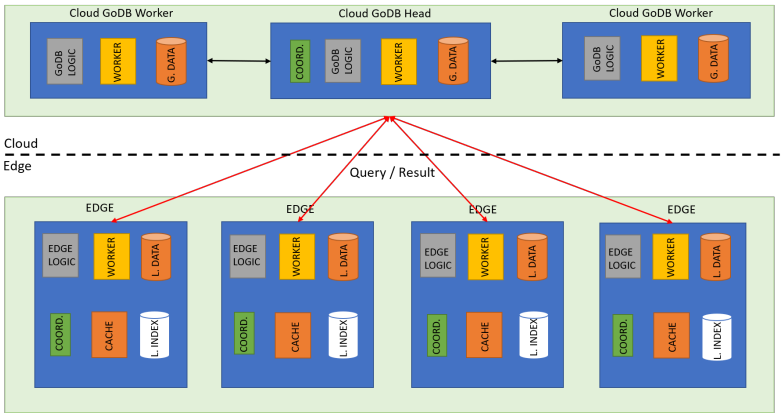
However, it lacks a declarative query model and graph partitioning technique to support heterogenous devices and targets only commodity clusters.

It is apparent that there are no graph querying system that solves the exact problem of graph querying in edge-cloud layout.

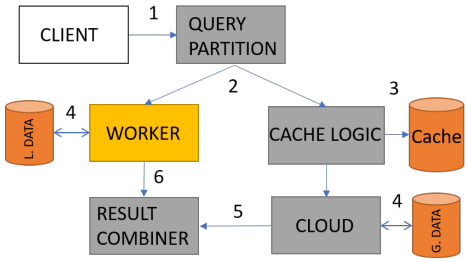
IV. Problem Definition and Approach

The proposed graph engine consists of different modules spanning across the edge and cloud layer. *Figure 2* illustrates the high level architecture of the system. Additional details and functionalities present in each layer is given below,

- **Cloud Layer:** This layer consists of multiple nodes (VMs) that are interconnected. One of these nodes is a the Head or master node while the other nodes are the workers. GoDB database engine is deployed on top of these nodes and serve as the core graph querying engine. These nodes are commodity grade. The global knowledge graph is hosted in the GoFS data storage
- **Edge Layer:** Edge layer consists of three essential modules. There is a edge logic module that includes the logic for query partitioning and result combiner. The coordination module is used to manage the edge which includes edge-cloud communication and communicate with other edges.
  The Cache module consists of statistics manager for maintaining the statistics, cache replacement logic and cache store for storing the queries and their corresponding results. It also performs cache check for any incoming query from the edge.



(a) High Level Architecture



(b) Logical Flow

Fig. 2: Proposed Design

It has a local data store which stores the local knowledge graph. This partition of graph is assumed to queried most frequently by the corresponding edge. Each edge consists of a worker which performs graph processing locally.

Figure 2 shows the logical flow for a single query from query submission to obtaining the results. A detailed description of each step is given below,

1) The graph query in declarative language is submitted to the edge from a client. The client could very well be the edge itself. Cloud level clients are not covered in this project
2) The query partition logic essentially splits the input query into two. One part of the query is run on the edge using the local knowledge graph and the other part is sent to the cloud layer. The objective of this partitioning is to run the query in parallel on both edge and cloud and merge the results
3) The cache logic receives the global query from edge and checks the cache first for a hit. It will also send the query to neighbouring edges to check for cache hit to allow for spatial locality. If there is a cache hit, the logic retrieves the query result or else it contacts the cloud layer to retrieve the result. Only global query result is cached as they provide the maximum benefit.
4) The edge worker receives the local query and start the processing for the given query type. The cloud head node receives the global query and computes the result using the worker nodes in parallel. The worker follows subgraph centric model similar to GoDB and the entire local knowledge graph is treated as a single subgraph for this purpose.
5) Once the result is computed in cloud, it is sent to result combiner in edge. Note that this is only a partial result for the original input query
6) The local worker on the edge sends the query result to the combiner. The combiner logic then merges the result from edge and cloud using a query type specific logic and sends the merge result to the client

The graph partitioning between edges and cloud happens by specifying a central vertex for each edge device. The central vertex along with its neighbours within a specified distance is captured and stored as the local knowledge graph. Adaptive partitioning will also be explored where partitioning is done based on query workload.

For the purpose of this project, the knowledge graph is assumed to be static. For dynamic graphs, additional logic is required in edge and cloud to update the indexes, cache and the graph data itself.

Also, depending upon the feasibility of deploying GoDB on the cloud, it might be necessary to replace it with other distributed graph database such as *Titan* [14] or *Dgraph* [15]. This should not affect any functionality in the edge as long as the interface is made constant.

Major implementation effort is needed to complete the worker and edge logic modules as these will have different logic for each query type. Feasibility of porting some of the query planning and optimization logic from GoDB will be explored before the actual implementation.

The implementation in edge layer will be done in Python language as it is known to be lightweight in terms resource requirement. The communication between the devices is handled through gRPC framework.

Cache data in edge layer will be in-memory as the cache size is generally small enough to fit in memory and will be spilled to persistent storage only if necessary.

Graph indexing is considered as good to have functionality with respect to the scope of this project. It will be potentially stored as a separate data structure and additional modules will be added in edge layer to make use of the indexes. This will accelerate the performance of local query execution and can help if the local query processing takes more time than its counterpart.

Additional details regarding the query partition and result combining logic is omitted due to lack of space. It will be added in depth in the final project report.

V. Proposed 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 [16] 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 [17] 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.

VI. Milestones

The following are the potential deliverables for mid-term evaluation,

1) Knowledge Graph partitioning will be implemented on the edge so that a local graph is determined and persisted on the edge
2) Query Partitioning in the edge layer will be implemented. Given a input query, it will be divided into local and global query based on the knowledge graph partition
3) Edge worker will be implemented which will run local queries on the local graph data
4) Result Combiner will be implemented to combine results of local and global query evaluation
5) All the above will be done for Vertex Search, Edge Search and Reachability query types
6) Relevant experiments for the above items will be performed as per the setup described in the previous section

7) Mid-Term report consisting of design, experiments and analysis will be submitted along with code

The following are the deliverables for final evaluation,

1) Any Feedback from mid-term evaluation will be incorporated into the project
2) Efficient caching (storing) of global queries and results in-memory on the edge device will be implemented.
3) Statistics manager and cache replacement logic for caching will be implemented on the edge
4) Support for Single Source Shortest path (SSSP) query will be added on the edge
5) Indexing for Vertex search, Edge search and Reachability will be implemented on the edge
6) Remaining Experiments will be performed as described in the previous section
7) Final report with full conceptual design, implementation details, experiments, plots and detailed analysis will be submitted with code

The following are the stretch goals for the project. These are definite goals and will only be accomplished as per availability of time.

1) Support for Single Source Shortest path (SSSP) query indexing in edge and cloud
2) Support for Subgraph isomorphism queries in both edge and cloud
3) Support for updating the knowledge graph from edge. (Addition, Deletion and Updating of vertices and edges)

References

[1] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A core of semantic knowledge," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 697–706. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242667

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[3] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824077

[4] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2484425.2484427

[5] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516. [Online]. Available: http://doi.acm.org/10.1145/2463676.2467799

[6] J. Wang, N. Ntarmos, and P. Triantafillou, "Graphcache: A caching system for graph queries," pp. 13–24, March 2017. [Online]. Available: http://eprints.gla.ac.uk/130141/

[7] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *22nd International Conference on Data Engineering (ICDE'06)*, April 2006, pp. 38–38.

[8] W. Fan, X. Wang, and Y. Wu, "Answering graph pattern queries using views," in *2014 IEEE 30th International Conference on Data Engineering*, March 2014, pp. 184–195.

[9] S. Seufert, A. Anand, S. Bedathur, and G. Weikum, "Ferrari: Flexible and efficient reachability range assignment for graph indexing," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, April 2013, pp. 1009–1020.

[10] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 1876–1888, Aug. 2018. [Online]. Available: https://doi.org/10.14778/3229863.3229874

[11] N. Jamadagni and Y. Simmhan, "Godb: From batch processing to distributed querying over property graphs," in *IEEE CCGRID*, 2016.

[12] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A subgraph centric framework for large-scale graph analytics," in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 451–462.

[13] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng, "A general-purpose query-centric framework for querying big graphs," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 564–575, Mar. 2016. [Online]. Available: http://dx.doi.org/10.14778/2904483.2904488

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

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

[16] 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

[17] 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