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 developed which can natively scale across edge and cloud devices. It supports declarative query model for vertex, edge, reachability and path queries. The engine has a novel caching 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 graph.**

I. Introduction

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. 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 that exists on the current edge-cloud layout. The proposed graph querying engine uses Tinkergraph 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 Caching:** A novel graph caching approach is used which caches a portion of 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 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 has a FIFO cache policy and can quickly match an incoming query with cache.

II. Novelty/Motivation

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.

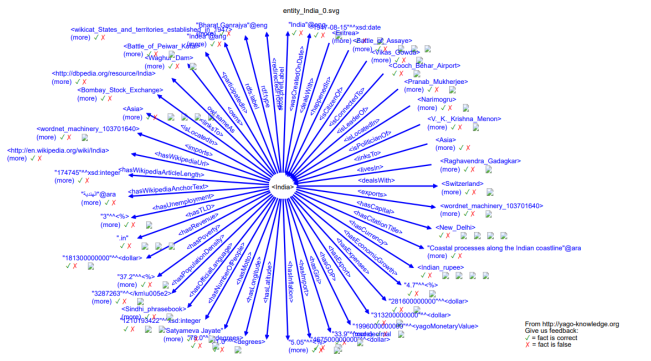


Fig. 1: Local Knowledge Graph

IV. System Design and Implementation

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

A. Remote (Cloud) Layer

This layer consists of an in-memory graph database engine *TinkerGraph* [14] running on the head node of *Rigel* Cluster. Integration of *GoDB* [11] 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. It is built on top of *TinkerPop* framework and *Gremlin* [15] programming language 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. However, using the same engine as cloud reduces the complexity of merging th query results and reduces semantic mismatches between the two layer. The *TinkerGraph* engine consumer less memory which can easily fit into the edge device.

2) Knowledge Graph Caching: The logic for caching 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. 1 shows a snapshot example of local knowledge graph centered around <India>.

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.
- *Reachability* - The reachability 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.*
- *Path Query* - A path query consists of n vertex predicates and $n-1$ edge predicates. Matching of Labels is considered as predicates in this work. The engine tries to find a path in the knowledge graph which matches the given path query. *null* in any section of the path query indicates a wildcard match for all possible values. In this case, if the length of path is less than the hops used for graph caching, local query is fired. However, if the length of query is greater, then only remote query is fired which is guaranteed to give all possible matching paths.

Some example queries of each type is given below,

```
{ // Vertex Search
"type": "vertex_search",
"filter": {
"has_label": "50_Cent",
"has": null,
"from": null,
"to": null }

{ // Edge Search
"type": "edge_search",
"filter": {
"has_label": "exports",
"from": null,
"to": null }

{ // Reachability
"type": "reachability",
"filter": {
"from": "India",
"to": "Barack_Obama" }

{ // Path Query
"type": "path_search",
"filter": {
"vertices": ["Mahatma_Gandhi", null,
             "Subhash_Agarwal"],
"edges": ["isCitizenOf", null]
 } }
```

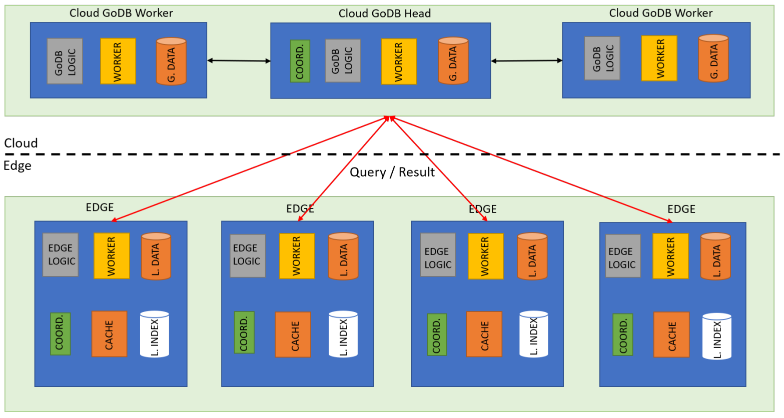
4) Combining Query Results: Combining the query results is implemented as follows,

- *Vertex & Edge Search* - The result set from local and remote server search are combined using *set union* operation
- *Reachability* - 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
- *Path Query* - The result from both local and remote server are combined as set union and returned as output. There is no need to merge individual results since the graph is not partitioned.

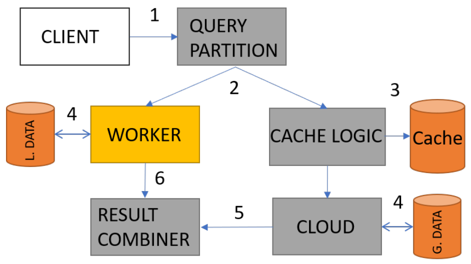
5) Remote Query Result Caching: It is expensive to query the remote server considering the network latency and so a cache module is implemented in the edge device. It leverages the temporal locality of the queries being fired from the edge. A FIFO (First-In-First-Out) cache is implemented which stores the results returned by remote server against the corresponding remote query. Whenever, a remote query is issued, it is checked for a cache hit and the result is returned from the cache if available. In case of cache miss, the result is fetched from remote server and added to the cache. Note that, local query results are not cached. A maximum cache size in terms of bytes is given as part of configuration and is used to determine when items from cache need to be evicted.

It has to be noted that all custom modules in edge layer were built in *Python* and queries were submitted to local and remote servers through *gremlinpython* package. No indexes were used in this mid term project. Also, the queries to remote server can performed concurrently due to an issue with *gremlin.* So, only one local query and one remote query can run concurrently at a time.

Figure 2 shows the logical flow for a single query from query submission to obtaining the results.



(a) High Level Architecture



(b) Logical Flow

Fig. 2: System Design

V. Experiments

The experimental setup is as follows: A single node in-memory version of *TinkerGraph* was run in the head node of *Rigel* cluster. The node has the following specs: 32 core AMD Opteron 6376 processor with 128 GB DDR3 RAM running CentOS 7 version. It is connected through 1 GBps ethernet and has 6 TB HDD and 256 GB SSD storage. Persistence was turned off in this layer.

A single edge device was spawned as Docker container with 4 cores, 1GB RAM and Ubuntu 18.04 LTS simulating a Raspberry Pi3B+ class of device. The network connectivity has a latency of 5ms and bandwidth of 100 MBps.

A smaller version of the *YAGO* [1] dataset was used in these experiments. It consists of 18845 edges, 14977 vertices and 1400 total no. of vertex attributes. The vertices consists of most popular entities in the knowledge graph.

There were some technical issues in converting the large dataset into a compatible version for *TinkerGraph*. This will be explored as part of final term project. The local knowledge graph was centered at <India> with 2 hop neighbourhood.

A. Vertex Search

Random queries with different values for the search fields uniformly distributed were generated for 500s and the time taken for local and remote query response was measured. 3 shows the time taken distribution for each query type with X-axis denoting the query type and Y-axis denoting the time taken in ms (log). It can be observed that the time taken by local queries are generally less compared to remote queries (approx. 5x). This is as expected since there is a network latency and size of remote knowledge graph slowing the remote server query time. The time taken is consistent for all query types under a given server. This is as expected since the query requires maximum of 1 hop distance traversal.

B. Edge Search

Similar to Vertex search, random queries were generated with different values in search fields uniformly for 500s. 4

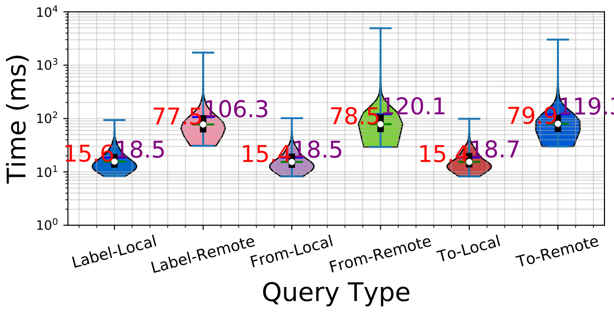


Fig. 3: Vertex Search - Performance

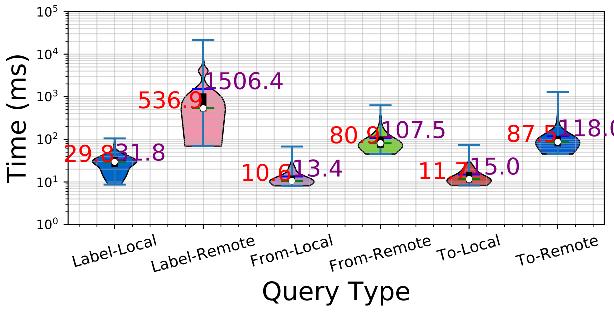


Fig. 4: Edge Search - Performance

shows the time taken in ms for different query types. The remote server query times are order of magnitudes higher than that of local which is as expected. The label search takes longest time since there are many edges having the same label unlike vertex where labels are unique. From and To searches will hit only a single vertex and its one hop neighbour and so their performance is similar to that of vertex search

C. Reachability

For reachability, the source vertex was fixed at <India> and the destination vertex was randomly assigned from the list of all vertices. The experiment was run for 1000s

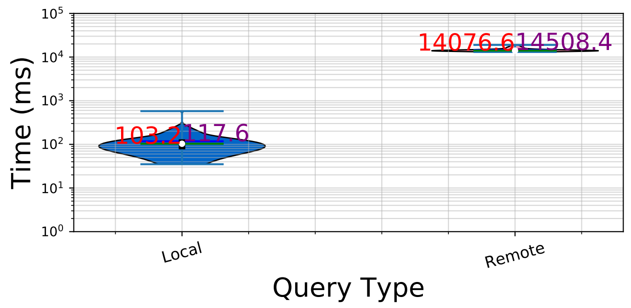


Fig. 5: Reachability - Performance

to allow for more data points. The no. of *cut vertices* was limited to 10 so that experiments complete in reasonable time. This means there could be some incorrect results. 5 shows the time taken in ms for different query types. Remote server took much larger time to respond since a series of queries were fired (one for each *cut vertex*). The time taken also depended on the length of paths returned in the result.

Other experiments to study the performance of path queries and cache is to be performed using Google Cloud Platform. These are not included as part of this report.

VI. Challenges

The major challenge in this project is integrating an existing graph engine. Considerable time was spent on making *GoDB* work in the cluster but eventually failed. Then focus shifted to *TinkerGraph* which had good documentation for querying. However, it was a challenge to setup a distributed environment for *TinkerGraph* with the help of documentation.

Availability of large resources was also a concern. For the initial part of the project, lab cluster was available but for the final set of experiments, cloud based resources had to be used due to non-availability of lab resources and other logistic challenges in terms of getting privileged access, required software etc. It would be better to provide the course participants with virtual machines for experimentation.

The project faced quite a bit of challenge in loading the *YAGO* dataset. It was available in *TSV* and *TTL* format. However, all of the graph database engines did not natively support them. It was a struggle to perform ETL for the dataset to convert into suitable format due non-availability of good libraries and large machines to handle the data. This resulted in wastage of time exploring different databases and trying to load the dataset.

VII. Conclusions and Future Work

Distributed Knowledge graph querying system was implemented and basic experiments were performed to evaluate its performance. Different optimizations like query result caching, graph caching, query partitioning were implemented to improve the performance.

As part of future work, indexing techniques will be leveraged to index the graph data in edge which can accelerate the local query performance. There are ample machine learning techniques that can be applied to knowledge graph to get useful insights. Distributed computation of these machine learning algorithms will be explored and integrated with the system.

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] "Tinkergraph." [Online]. Available: http://tinkerpop.apache.org/

[15] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th Symposium on Database Programming Languages*, ser. DBPL 2015. New York, NY, USA: ACM, 2015, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2815072.2815073

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

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

[18] 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

[19] 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