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* [1] running on the head node of *Rigel* Cluster. Integration of *GoDB* [2] 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* [3] 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.

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

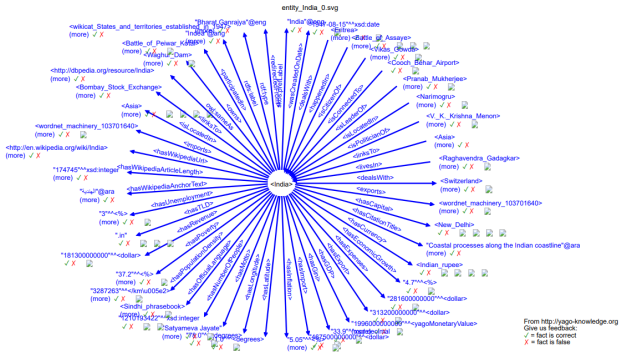

Fig. 1: Local Knowledge Graph

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
- *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*.

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
    }
}
{ // Path Search
    "type": "path_search",
    "filter": {
        "from": "<India>",
        "to": "<Barack_Obama>"
    }
}
```

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

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.

II. 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.

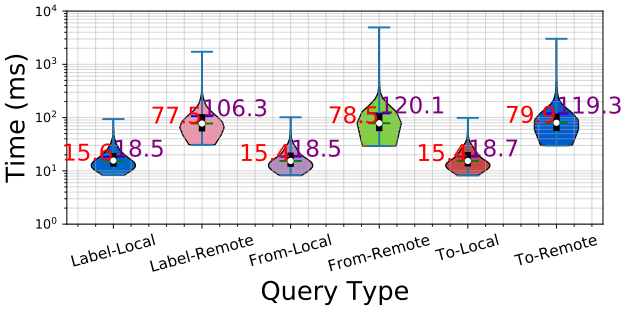

Fig. 2: Vertex Search - Performance

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* [4] 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. 2 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. 3 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

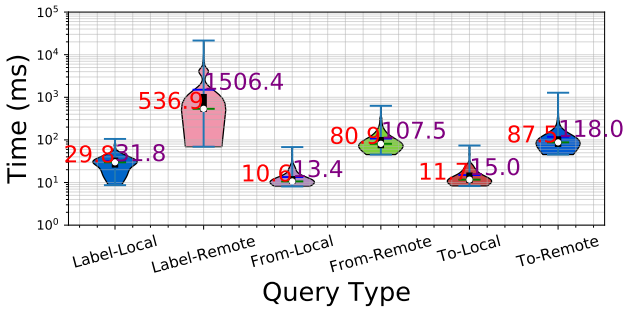

Fig. 3: Edge Search - Performance

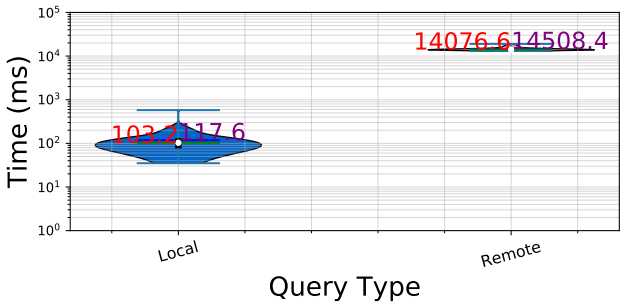

Fig. 4: Path Search - Performance

C. Path Search

For Path search, 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 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. 4 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.

III. Conclusion

Regarding the mid-term deliverables, key modules to make edge layer work were implemented successfully. However, these modules have to be extended and refined to handle different other query types and possible corner cases. One of the major item is missed in this deliverable is the non-availability of experiments using the full dataset which could clearly explain the scalability of the system.

References

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

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

[3] 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

[4] 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