Chapter 1

Graph Paradigm - Microsft GraphEngine

Contributed by Famim Talukder

1.1 Background

Large graphs are very common in numerous different domains, such as social networks, communication networks and biological protein-protein-interaction networks (PPI), etc [1]. Efficiently programming, managing and processing such graphs have posed a real problem, especially for such large graphs [4]. The needs of these networks can be categorized into two main categories - online query processing, for which low latency is required, and offline graph analytics, which require high throughput. An example of online graph processing is deciding if there is a path between two people in a social network, while computing centrality measures for PPI networks will generally be done in offline graph processing. Offline analytics are usually executed in batch mode, where large portions of the graph is processed in parallel. Keeping the graph data in main memory improves the performance significantly, as such processing often requires repeated and efficient access to graph data.

Despite the different applications of graph data, graph computations usually have some similar characteristics. One similar characteristics of these graphs is that the processing time of these graphs are dependent on the I/O access time, since large amount of data needs to be accessed for graph computation. Moreover, graph processing often requires a high degree of random access data. Random access is especially significant for graph exploration queries like the shortest-path, DFS, etc. However, even with modern storage devices, data bandwidth and random access is still not proficient enough [5]. Storing data in main memory offers significant advantages over traditional disks or flash storage.

1.2 Expressing Graphs

GraphEngine is designed to provide both fast online query processing, as well as, efficient offline graph analytic. To achieve this, GraphEngine store graphs in-memory, with a strongly-typed RAM store, in combination with a general purpose distributed computation engine, allowing GraphEngine to support low latency online query processing, while simultaneously offering high throughput offline graph analytics.

GraphEngine stores graph in-memory, on the well connected servers, and access them using high-speed networks. The motivation behind this is that high-speed networks are becoming readily available while DRAM prices are decreasing, which will lead to inexpensive DRAM solutions in the very near future. Nonetheless, GraphEngine does not have any specific hardware or software dependencies, and can be adapted to different types of graphs.

To be able to service these graphs, GraphEngine does not include a comprehensive list of built-in computation modules, like most other systems [2, 3]. However, existing GraphEngine modules can be modified easily to support graph computation. This design allows GraphEngine to efficiently support different types of graph using its fast graph exploration methods, the central part to its superior performance.

1.2.1 GraphEngine Components

Distributed graph mode is main utilized mode with GraphEngine. This mode allows GraphEngine to be deployed on numerous different machines, and require a number of system components to communicate through a network. Each component has a specific role, and is explained in details below.

Server

A server has two main roles. It is tasked with storing a portion of the graph data in-memory. Moreover, a server is also in charge of graph computation, which usually involves sending messages and receiving messages from other GraphEngine components.

Proxy

Proxies only handle messages without storing any data. They serve as an intermediary between clients and servers. For example, they can serve as data aggregator, receiving requests from clients and relaying them to appropriate servers. Once the results of the requests are received, they might be aggregated by the proxies before relaying it back to the clients. As a result, they are optional and can be added to the system as required.

Client

Clients in GraphEngine communicates with GraphEngine clusters. It serves as the user-interface between GraphEngine and the end users. Client-side logic is implemented as a shared library, allowing processes to store and load results of the computations done by the GraphEngine servers. Communication between clients and other components is achieved by utilizing the GraphEngine API.

1.2.2 Memory Cloud Abstraction

GraphEngine uses numerous machines with a globally addressable space, called the memory cloud to store graph data. Each of the machine's local memory is also partitioned in separate sections, called memory trunks. The partition of the local memory enable trunk-level parallelism without locking overhead. Moreover, the partitions decrease hashing conflicts, making the system even more efficient. As a result, there are a total of 2^P addressable memory trunks, and $2^P > m$, where m is the number of machines. Memory trunks are also backed up on a shared distributed file system, Trinity File System (TFS), to support data persistence, and fault tolerance.

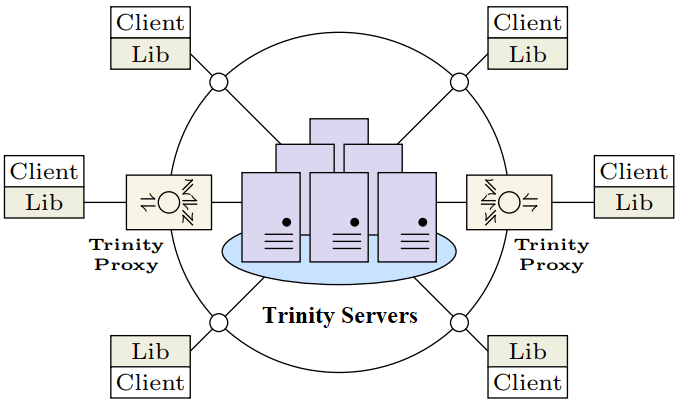

Figure 1.1: GraphEngine, previously named Trinity, cluster structure with the interaction of the different components.

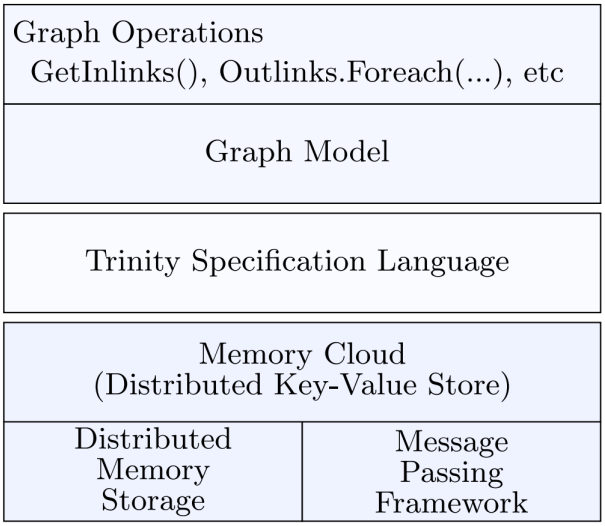

Figure 1.2: The different System layers of GraphEngine.

A distributed key-value store is built on the memory cloud, as shown in Figure 1.2. The distributed memory storage is responsible for managing memory and providing mechanisms for concurrency control on the GraphEngine servers. On the other hand, the message passing framework provides an efficient, one-sided, machine-to-machine message passing infrastructure.

GraphEngine also uses a globally shared addressing table, since a centralized adressing table implementation is prone to failure. However, copying the table for each machine leads to inconsistency, as well as performance issues. As a result, a primary replica is maintained on a leader server machine, and TFS keeps a persistent copy of the table. Updates are made to the primary table, as well as the persistent copies, before they are committed.

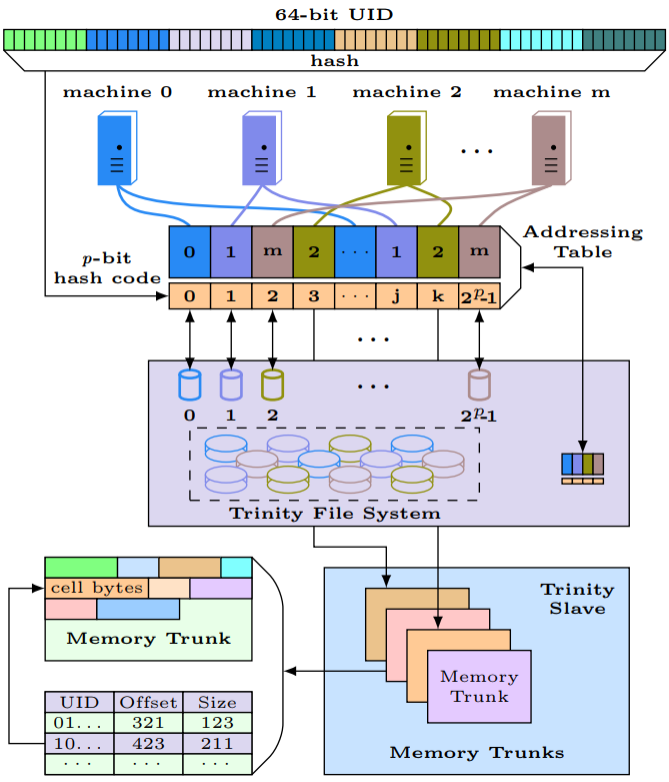

Figure 1.3: Key-value data partition, addressing and look up.

1.3 Syntax

GraphEngine processes graphs by first storing them in-memory on a distributed memory cloud, using a key-value pair. A schema is defined using TSL. When a value is associated with a schema, it becomes a cell, and the key-value pair becomes the (cellId, cell) pair. Cell are used to implement nodes in the graphs. Cells can also contain different information, for instance in undirected graphs, cells can contain a set of cellIds for its neighboring nodes. Cells might also contain metadata, like its name, description, etc. In directed graphs, the cell's set of neighboring nodes will represent outgoing edges. By using this paradigm, GraphEngine can generally avoid explicitly storing the edges. If a need to explicitly store edges, then (cellId, associatedData) key-value pair can be used to store edges. Key-value pair for edges allow GraphEngine to store complicated graphs, like hypergraphs.

1.3.1 Trinity Specification Language

Trinity Specification Language (TSL) is a high-level language, similar to the popular `C` languages, used to model data and network communication. TSL allow GraphEngine to adapt to different graphs, provide a programmable environment, and service their specific needs. TSL defines a schema, which is used to handle blob values stored in the key-value pair.

Figure 1.4 model two different types of graph nodes, Movie and Actor, using the `cell struct Movie` and the `cell struct Actor`. `Cell structs` are basic data types used to define graphs and can contain an arbitrary number of primitive data types, data containers, or user-defined structs. Edge types are also defined in the script with `EdgeType`, in this example the edges are modeled implicitly as simple edges. Edges can be explicitly defined with a few modifications to the `EdgeType`.

```
[CellType: NodeCell]
cell struct Movie
{
    string Name;
    [EdgeType: SimpleEdge, ReferencedCell: Actor]
    List<long> Actors;
}
[CellType: NodeCell]
cell struct Actor
{
    string Name;
    [EdgeType: SimpleEdge, ReferencedCell: Movie]
    List<long> Movies;
}
```

Figure 1.4: Modeling a Movie - Actor graph. The syntax is very similar to the `C` languages.

TSL can also model network communication, using its own message passing strategy. TSL provides supports for both one-sided communication, as well as, bulk-synchronous message passing, and transparent message passing to increase network throughput. Basically, TSL can pack multiple message in one packet to decrease message passing overhead.

```
struct MyMessage
{
    string Text;
}
protocol Echo
{
    Type: Syn;
    Request: MyMessage;
    Response: MyMessage;
}
```

Figure 1.5: Message passing using TSL.

Figure 1.5 shows an example of a simple echo protocol where a client sends a message to the server and the server relays the message back. In this example, the message uses asynchronous message passing. TSL will create a message handle during compilation, and the user will need to implement logic for the handler. GraphEngine will manage the message dispatching, packing, etc. for the user.

1.4 Key Graph Primitives

Discuss here what are the key graph primitives supported by the paradigm.

1.5 Execution Model

There are several different uses for GraphEngine, hence there are different execution model for each program. Each execution model offers its own advantages.

1.5.1 Traversal Based Online Queries

Many application require a traversal based query. For example, people search on a social network requires very fast graph exploration algorithm to determine relationships. For example, for a given

user, find anyone named 'David' among his/her friends, his/her friends' friends, and his/her friends' friends' friend. This is a very practical query, and is very common in social networks. These queries cannot be predicted, nor pre-computed due to the dynamic nature of the queries and the social networks.

GraphEngine tackle this problem by leveraging its efficient memory-based graph exploration capabilities. In a simulation using a synthetic power-law graph with 800 million nodes and 104 billion edges, comparable to Facebook network in 2013, and with only eight-machine cluster, GraphEngine is able to explore the graph in less 100 milliseconds on average. Specifically, GraphEngine can explore $130 + 130^2 + 130^3 \approx 2.2$ million nodes in less than one-tenth of a second. GraphEngine is able to achieve this performance because it sends asynchronous requests recursively to remote machines, which is designed to efficiently handle memory access and optimize its network communication.

1.5.2 New Paradigm for Online Queries

Since GraphEngine stores graph data in-memory, GraphEngine can support numerous complicated queries not supported before. For instance, sub-graph matching is a difficult problem which required pre-computed indices which requires super-linear build time, and also perform computations in super-linear space and time as well. As a result, such a computation is very inefficient with existing systems. However, GraphEngine can use its fast random data access and parallel computation to perform sub-graph matching on large graphs quickly. Specifically, it can perform sub-graph matching on a 128 million nodes, with an average node degree of 16, in less than one second. Moreover, it also enables a global view of the graph, which can improve performance of many algorithms.

1.5.3 Vertex Centric Offline Analytics

GraphEngine also implements the traditional vertex centric model for offline graph analytics. In this model, each vertex acts as an independent agent. After several iterative tasks, there is a designated super-step, where a communication task is expressed. During the super-step, the communication and computation is done by each vertex independently of each other. However, in GraphEngine, communication for each vertex is limited to its neighbors, unlike other models [3]. Consequently, a message can only be received by a neighboring node and can only be sent to a neighboring node. GraphEngine places a restriction on the vertex centric model because it allows GraphEngine to take advantage of this restriction to optimize performance. Since messages can only be sent to a set of nodes by any neighboring node, messages can be prepared in advance before computations are executed on nodes, as shown in Figure 1.6.

Ideally, messages will need to be passed within a local machine, allowing the jobs to run without waiting on others. However, optimizing partitioning of a graph by disconnecting the minimum number of edges is, itself, a difficult task. In GraphEngine, the nodes are divided into two categories - hubs and everything else. Essentially, the high degree nodes are placed in one category and the remaining nodes in another. Messages are buffered from the first category at each iteration, essentially not participating in Figure 1.6b partition, since they will be sending messages to most of the other nodes. By taking advantage of such partitioning, GraphEngine is able to achieve significant performance improvements.

1.5.4 New Paradigm for Offline Queries

Trinity also supports probabilistic inference to determine query results for the whole graph, based on the partition on a single machine. This is more effective when the graph is randomly partitioned,

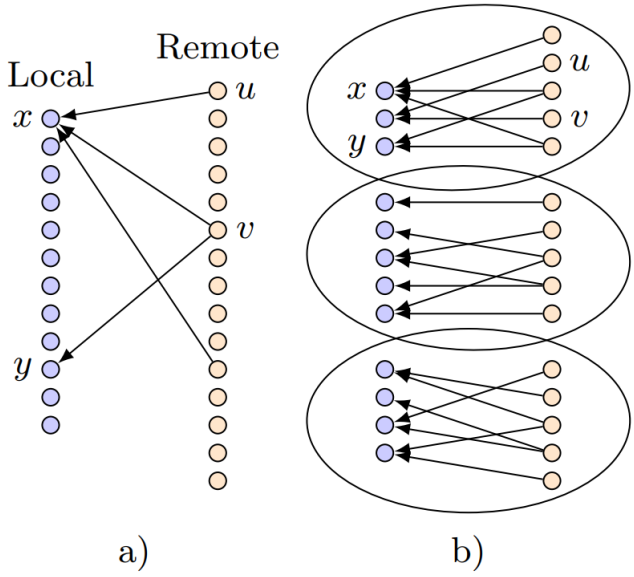

Figure 1.6: Communication view of nodes on a local machine.

hence a partition is representative of the whole graph. Utilizing such methods can significantly reduce network communication overhead, but it only returns approximations of the results. However, simulations on synthetic networks have been very promising in this regard.

1.6 Examples

Discuss here any code examples and/or performance reports

1.7 Conclusion

This paper discusses the advantages of using GraphEngine, created by Microsoft Asia. GraphEngine is an in-memory distributed graph paradigm with an efficient computation engine. It services quick online graph queries as well as full offline graph analytics. The distributed storage system is designed efficiently for random data access, allowing for quick graph algorithms. The advantage of GraphEngine is its scalability and performance on large graphs, which is orders of magnitude better than most other systems.

Bibliography

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[2] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[3] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[4] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. Horton: Online query execution engine for large distributed graphs. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1289–1292. IEEE, 2012.

[5] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.