GoFFISH System Architecture

The USC Graph oriented Framework for foresight and insight *using Scalable Heuristics* is an open source stack that provides various components, ranging from storage and resource management to a high level programming model, specifically designed for analytics on large time-series graph data sets.

# Introduction and Background

The proliferation of ubiquitous physical and virtual agents that sense, monitor and track human and environmental activity, data is streaming more continuously and is intrinsically interconnected.

Two defining characteristics of such datasets endemic to both the Internet of Things and Social Networks are the temporal attributes and the lateral relationships that exist between them.

GoFFISH has been designed with the goal of supporting large scale analytics on such data sets, we call ***time-series graphs***. This requires not only efficient storage and retrieval but also warrants special attention on the application composition model that will allow users to easily define and execute large scale algorithms/applications on these data sets.

## Time-Series Graphs

Time-Series graphs in general can be defined as the graph that varies in structure (vertices and/or edges) as well as values associated with the graph elements. However, the current GoFFISH system only supports a subset of time-series graphs where the structure of the graph remains constant and only the values associated with the graph elements (vertices and the edges) change over time.

Figure 1 shows such a time-series graph. These restricted Time-Series graphs consists of one ***Graph Template*** that defines the graph structure and any static or default values associated with the graph elements and one or more ***Graph Instances*** that define the values for the graph elements at different time intervals. For simplicity, we assume that at each time interval, values for all the elements are well defined.

Figure 1. Time Series Graphs

The fixed graph structure allows us to employ static partitioning of the graph template and use the same scheme for all other instances. This allows several optimizations as described later.

## Subgraph-Centric Programming Model

Vertex-Centric programming model with Barrier synchronization (BSP) such as Pregel (and Giraph) have been empirically proven to be more efficient than the traditional graph algorithms mainly due to intrinsic ability to parallelize the algorithm by executing a "worker" at each of the vertex in parallel during a BSP super step.

We propose a subgraph-centric programming model, an extension to the vertex-centric model. In the proposed model, a graph (or a partition, if distributed) is divided into a number of connected components (weakly or strongly connected, based on the intended application). A single "worker" is executed for each of these components, possibly in parallel if enough resources are available. Unlike the vertex centric approach in which the worker only computes and updates the state of a single vertex, in our model, the worker can leverage the information that the given component is guaranteed to be connected and apply a more sophisticated process during each superstep.

We envision that developing algorithms using the subgraph centric BSP abstractions will have multiple advantages over the vertex centric approach, viz., reduction in communication overhead, reduction in process overhead due to reduction in the number of workers, and in some cases reduction in the number of supersteps as well.

Figure 2 Subgraph Centric Programming Model

# GoFFISH Components

Figure 1 shows various components of the GoFFISH stack and are described below.

Figure 1. Components of GoFFISH Software stack

* **GoFS**: It provides a distributed data storage for large graphs. It is specifically designed to store and efficiently retrieve large time-series graph data sets which consist of one graph template and a number of graph instances (typically, spread over a period of time). *(See section X)*
* **Floe**: This is a generic continuous dataflow engine that provides a scalable and resilient execution for dataflow applications. The discussion of Floe is out of scope of this document and interested readers are referred to the [technical report](file:///C:\Users\Administrator\Documents\PUT A LINK HERE). However, we briefly describe the two components of Floe with which we directly interact.
  + ***Resource Management Plugin***: Floe provides a plug-in architecture for its various components, including resource management. We leverage this to implement a GoFS aware resource management plugin. *(See section XX)*
  + ***Distributed Continuous Dataflow Engine***: Floe allows building large scale continuous dataflow by supporting various dataflow patterns to govern execution of processing elements as well as data transfer between them. We leverage these patterns to implement a generic Bulk Synchronous Parallel (BSP) abstraction over the Floe engine. *(See Section XXX)*
* **Gopher**: Gopher provides a “subgraph-centric” programming abstraction and execution framework for running analytics on large time-series graphs. This is built using the BSP abstraction built on top of the Floe engine. (See Section XXXX)

# GoFS Architecture

GoFS is a distributed storage system optimized for storing and retrieving time-series graph data sets. We first describe the interactions between various entities (Figure 2) in the system followed by a detailed description of the storage layer on the data nodes.

Figure 2: Interactions between *Client, Name Node* and *Data Nodes*

The system consists of a Name Node and a set of Data Nodes. The Name Node is responsible for storing and maintaining a list of stored graphs and the mapping from each partition of the graph to their physical location. The Data Node is responsible for storing each of these partitions and provides APIs to access local partitions.

Following steps are taken to store a time-series graph into the system.

1. Client partitions the Graph template using an acceptable Graph partition algorithm.
2. Client submits a request to the name node. R\_i = {GraphID, Number of Partitions} which returns a list of data nodes to store each of the partition.
   1. The client then identifies subgraphs (weakly connected components) in each of the partitions
   2. Creates a template metadata slice for each partition (see section YY)
   3. serializes each slice using preferred serialization method and
   4. finally sends it to the data nodes for storage.
3. At the end of Step 3 stage the Client also communicates with all the data nodes and notifies the mapping between subgrah ids to partition ids for each subgraph as well as a mapping between remote vertices and the subgrap id. This enables each subgraph to communicate with the remote vertices using this map.
4. The client then partitions each of the graph instances using the partition info obtained from the template partitioning step.
5. It then creates a number of slices optimized for storage size as well as retrieval latency for each of the subgrah instances which are then stored in the data nodes matching the corresponding template partitions. (see section ZZ)

# GoFS Partition Storage Optimizations

GoFS Partition storage on each of the data nodes is optimized for not only disk storage but also to allow efficient read of selective subset of the data as queried by the user in terms of both the temporal range as well as a subset of values associated with the graph elements.

The approach to perform such optimizations is to create a number of blocks (slices) each of which contains a subset of the graph data. Each slice is serialized to a binary format to optimize for disk space. Also, each slice is read or written at once from the disk to optimize for read/write speed.

Figure 5. GoFS Storage Layer (API and Slices)

## Partition Slices

Each data node stores one (or more) partition of a time-series graph (i.e. a partition template and a number of partition instances) as shown in figure 5.

Following slices are created for each partition

|  |  |  |
| --- | --- | --- |
| **Partition Type** | **Description** | **Contents** |
| Partition Metadata Slice | This is used to store the mapping between the partition and various slices associated with that partition including the Partition Slice and various instances slices | TODO: Santosh/Soonil  po |
| Partition Slice | It is used to store the structure as well as static information about the graph partition. Entire partition is stored in a single Partition Metadata slice. |  |
| Instances Slice | It is used to store values for ***exactly one Property*** associated with either the vertex or the edge for one or more subgraphs.  A bin packing strategy is employed to decide which Subgraphs and how many instances should be put together into a single slice to maintain balance across various partition instance sclices |  |

## Serializers/DeSerializers

We support a pluggable model to deploy different serializers defined by the user. Currently, we support ***kryo and Java serializer*** to convert the slices into a compact binary format to be stored on the disk. However, we will continue research and development of specialized serialization techniques that allows better storage as well as read/write optimizations, especially designed for time-series graphs.

# Gopher Architecture

Figure 6 shows various components of the Gopher framework and are described below:

Figure 6. Gopher Architecture

Gopher is built on top of GoFS and the Floe dataflow processing engine. The former acts as a distributed graph storage and the latter provides basic framework for running gopher workers as well as a communication layer for message transfer between the workers.

## Gopher Components and Execution

Gopher is built as an application on top of the Floe engine. This implies that different components of Gopher are built as Floe pellets. Specifically there are two such components – Task Controller, and the BSP Processor

The Task controller acts as the entry point and a synchronizer entity to the gopher application. It is responsible for bootstrapping the application and transmitting any initial message that needs to be sent to each of the workers. It is also responsible for coordinating barrier synchronization and super step management for the gopher application.

The BSP Processor is responsible for executing the user logic of the algorithm. One task is executed for each subgraph in the graph per superstep. Depending on the partition mechanism used by GoFS, these subgraphs may be distributed across a number of machines in the cluster. For each machine, depending on available resources (memory and cpu), one or more of these tasks may run in parallel. Any remaining tasks are executed sequentially as previous tasks are completed on that machine. The Task controller is notified when each of these tasks are completed. Once all the tasks, across all machines, are completed, and the messages are transferred, the Task Controller can then choose to execute the next super step if required.

## Iterative BSP

In addition to the traditional BSP model, Gopher also supports an iterative BSP model. In this model, the user can choose to continue the algorithm for a number of iterations of a BSP algorithm without losing state or restarting the application. This is especially useful when working on data-sets such as time-series graphs where each of the iterations may work on different instances of the graph.