EAGr: Supporting Continuous Ego-centric Aggregate Queries over Large Dynamic Graphs - Summary

Gewen He
February 27, 2015

1 Problem Statement

1.1 Inputs

- Data graph model $G(V, E)$. Nodes represent communities, groups, user tags, web pages and so on. Edges can be friendships, memberships. As a result, some edges are bidirectional.
- A time stamped structure data stream $S_G$. (add/delete nodes/edges)
- For some node $v$, a content stream $S_v$. (all the updates on node $v$)

- Query model $Q(F, w, N, pred)$.
  - $F$ is a aggregation function. (i.e. sum, max, min, top-k and arbitrary aggregation functions)
  - $w$ denotes a sliding window over the content data streams. (i.e. $w=3$ means the 3 latest updates on a node $v$ who records the latest visited webpages.
  - $N$ denotes the neighborhood selection function. (i.e. $N$ can be $v$'s 2-hop neighborhood.)
  - $pred$ A subset of $V$ for which the aggregate must be computed. (i.e. We only want to do aggregation on some important nodes in $G$

Simply speaking, some nodes of the graph are updated in a period of time like a stream and we want to do some kinds of aggregation on some nodes of $G$ at any time stamp efficiently. Figure 1 is an example.

1.2 Outputs

Query results.

This paper aims to accelerate the query processing with the help of the index.

2 Motivation

- On-demand approach, where the neighborhood is traversed in response to a read, is unlikely to scale to the large graph sizes, and further, would have unacceptably high query latencies.
3 General framework

Given G and Q, we can have build an Overlay Graph. Overlay Graph of G is a "compressed" form of G’s equivalent bipartite graph, say A(G). We compress A(G) by finding bi-cliques in A(G) then for each bi-clique we replace the edges in A(G) in this bi-clique by adding a PartialAggregator that connect both sides of the current bi-clique.

\[ \text{Simply speaking, a bi-clique in a bipartite graph is a set of nodes in the write part, say } W(c), \text{ and a set of nodes in the read part, say } R(c), \text{ such that any node in } W(c) \text{ has an edge to any node in } R(c). \]

i.e. We have two bi-cliques in G. \( c_{PA1} \) is \((a, b, c), (e, g, f, c, d)\). \( c_{PA2} \) is \((d, e, f), (g, f, c, d, a, b)\)

For heuristic reason, this paper wants the number of edges in an OverlayGraph to be as small as possible. How to find Partial Aggregator will be discussed later.

After we constructed an overlay graph, we will make a data flow decision on each nodes. Simply speaking, we will assign each nodes in OverlayGraph a role- either a "PUSH" role or a "PULL" role. \( \text{How to decide the roles will be discussed later.} \)

When there is any updates, "PUSH" nodes are always updated with current Aggregation result. While for "PULL" nodes, we only update its Aggregation result when it will be used in read request from another node \( v \) upstream of it.

i.e. See figure 2. Roles of nodes are already decided. When any node in a,b,c are written in any time, we will first update a,b,c themselves if they are changed. Then since the aggregation on PA1 is definitely changed due to changes in a,b,c, we will update PA1 on time. Since e and f ’s aggregation result solely depends on PA1, they can be immediately updated as well. However g’s Aggregation also partially depends on PA2. Since PA2 is not a "PUSH" then we can’t update g’s Aggregation result up-to-date.
4 Constructing Overlay Graph

For any "read" node \( v \) in \( A(G) \) there will be a set of write nodes in "write" that are required in the Aggregation \( (N(v)) \). If we say this is a transaction \( v\{N(v)\} \) then we will have \( |\text{readnodes}| \) transaction. Now the problems is reduced to a frequent pattern mining problem. We want to find find some frequent items set such that they fulfill some requirements (due to we can have different aggregation functions) while the number of edges in OverlayGraph is minimized.

Typical Requirements:

1. Duplicate sensitive Aggregation functions. (Sum) For each pair of write-read nodes we can only have one path.
2. Duplicate insensitive Aggregation functions. (Max) For each pair of write-read nodes we can only have more than one path. (i.e. figure 4 (a) )
3. Aggregation function with negative inputs. (i.e. figure 4 (b) )
4. We can further reduce the number of edges in Overlaygraph by building multiple overlay. (i.e. figure 4 (d) )

How to find bi-cliques

1. Building the FP-tree. i.e. \( c_{w}\{a_{r}, e_{r}\} \) in figure 3(a) means we there are two transaction: reader \( a_{r} \) and \( e_{r} \) require this writer node \( c_{w}. \)
2. Mining bi-cliques. Any path from root to a node, say \( v \), means a bi-clique such that all the writer nodes in the path and the reader nodes in \( v \) form a bi-clique. i.e. Path: root-\( d_{w}-c_{w} \) means \( d_{w}, c_{w} \) and \( a_{r}, e_{r} \) form a bi-clique.
   We want to find a bi-clique s.t. Length of path * \( v \)'s support - Length of path - \( v \)'s supports is maximized since we want the OverlayGraph has as few edges as possible.
3. Remove all the edges in this bi-clique from \( A(G) \) then go to 1 (rebuild the FP-tree to find another bi-clique)
For very large graph:

1. Compute Shingles for each reader. Then group readers by Shingles. Use each group as input and find bi-cliques in each group.

2. Replace nodes and edges of $A(G)$ in each bi-clique with a virtual node ( aggregator). Repeat 1.

5 Dataflow Decision

Initially, all the write nodes in Overlay Graph will be set to "PUSH" for obvious reason. Another observation is if any nodes in OverlayGraph is labeled "PUSH" then all its upstream nodes have to also be "PUSH". The reason is we can’t node do an Aggregation if some of the inputs are not available. As a result there can not be an edge from a PULL to a PUSH

Under above constraints, we have an partition problem:

• Inputs:
1. The OverlayGraph $O$
2. Each node in $O$ has a cost of being "PULL" as well as a cost of being "PUSH".

Initially, Writer nodes are assigned a PUSH frequency, say $S_f$, as its number of updates and Reader nodes are assigned a PULL frequency, say $U_f$, as its number of times being queried. We can get the other nodes’ PUSH and PULL frequency by propagating from both sides.

\[ \text{i.e. } U_f \text{ of } i_3 = U_f \text{ of } (p_r+s_r) = 1+2=3; \quad U_f \text{ of } i_1 = U_f \text{ of } (m_r+n_r+i_3) = 4+3+3=10. \]

\[ S_f \text{ of } i_1 = S_f \text{ of } (a_w+b_w) = 3+2=5; \quad S_f \text{ of } i_2 = S_f \text{ of } (c_w+d_w) = 3+2=5; \quad S_f \text{ of } i_3 = S_f \text{ of } (i_1+i_2) = 5+5=10. \]

For different Aggregation functions, the cost of being PUSH should be the same as the PUSH frequency as if it is a PUSH node its aggregation value are always set up to date, we only need to fetch it. However the cost of being PULL Varies.

\[ \text{i.e. If } F \text{ is SUM then the cost of being PULL could be the degree of the node } \times \text{ its } U_f, \text{ since in order to compute the aggregation value we have to read } |\text{degree}| \text{ nodes in upstream whose } U_f \text{ is equal or larger than current } U_f. \]

\[ \text{If } F \text{ is MAX. The the cost of PULL could be expected to be } \log_2 |\text{degree}| \times U_f \]

- Outputs:

A partition of all the nodes in $O$, s.t. Each node either in PUSH or PULL and there is no edge from node in PULL to node in PUSH while the cost is minimized.

- Solution

The original problem is to solve \( \text{Min } \sum_{v \in \text{PUSH}} \text{PUSH}(u) + \sum_{v \in \text{PULL}} \text{PULL}(v) \) while we have edge direction constraints.

We build a new Graph $O'$ with all the nodes and edges of $O$ while setting the weight...
of all edges to $\infty$. Then we add two new nodes $s$ and $t$ and for each node in $O'$ if its PULL cost - PUSH cost $< 0$, then we add an edge from $s$ to $v$ while setting the weight of the edge as PULL cost - PULL cost; if its PULL cost - PUSH cost $> 0$, then we add an edge from $v$ to $t$ while setting the weight of the edge as PULL cost - PULL cost;

i.e. See figure 5. We add an edge from $s$ to $i_3$ with weight 4 since the PULL cost of $i_3$ is $2*3=6$ and the PUSH cost is 10.

![Diagram](image)

Figure 6: Cutting the edge from $s$ to $i_3$ is a MinCut. As a result, $i_3$ is no long a PULL node.

If there is a path from $s$ to $t$ then it means there is at least one edge from a node in the set of nodes reachable from $s$ to a node in the set of nodes that can reach $t$. We will find a minimum cut of edges in $O'$ to cut all the paths from $s$ to $t$. Then the set of nodes reachable from $s$ are the PULL nodes and the set of nodes that can reach $t$ are the PUSH nodes.

Why correct? Setting those nodes whose PULL cost is larger than PUSH cost to PUSH and vice-versa will necessarily have an overall minimum cost. By minimum cutting in $O'$ we dealt with the edge direction constraint and flip some node to the other partition at a minimum incremental cost.