Background

● Streaming Graphs
  ○ Super common - social networks, real-time traffic info, Web graph
● Processed via incremental algorithms
  ○ Tornado, Kineograph, Stinger, Naiad
● Why do we need KickStarter?
Other Streaming Graph Frameworks

- Does batch updates
- Maintains intermediate approximate results
- When a query arrives, start at the most recent intermediate result and do the rest of the calculations needed (in a branch loop)
- Makes sense: the values right before the updates are a better approximation of the actual results
Problems?

- Is that true? Will values right before an update actually be a good approximation of the actual results?
- Not necessarily - especially for edge deletions
- We deal with monotonic computations
  - Calculating some data to a vertex that only ever increases / only ever decreases
  - SSSP, BFS, Clique, Label Propogation
- Edge deletions break monotonicity - invalidate intermediate values

- (Edge deletions are common in real world graphs!)
• Breaking monotonicity can have two results:
  ○ Incorrect results
  ○ Poor performance

• Let’s try it out!

Figure 1: Three different scenarios with respect to the use of intermediate values after an edge update.
(b) After deleting the edge, the values are clearly incorrect

(c) After deleting the edge, try setting the value for D back to the initial value - however, still incorrect
Problem 2: Slowness with SSSP

- The deletion of the edge renders B and C disconnected from the rest of the graph.
- Each iteration will bump up the values of B and C - takes forever to reach the (correct) value of MAX.

![Diagram of a graph with vertices A, B, and C and an edge A→B.]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before deletion</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>A→B deleted</td>
<td>5</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>MAX</td>
<td>MAX</td>
</tr>
</tbody>
</table>

Figure 6: While using the intermediate value for vertex B yields the correct result, the computation can be very slow; the initial value at each vertex is a large number MAX.
When Incorrect? When Slow?

- Key difference is vertex update function
- In the first type, the update function only performs value selection
  - No computation is done
  - A value is propagated along a cycle
  - Incorrect result
- In the second type, the update function does some computation
  - Disallows cyclic propagation
  - Algorithm will eventually stabilize at the right value
KickStarter - How do we fix this problem?

- Edge additions are fine - edge deletions are dangerous
- Right after an execution is forked for an edge deletion, we add a trimming phase
- Unsafe vertex values are adjusted before feeding it into the forked execution
  - What is unsafe? Values that were dependent on the deleted edge

\[ \Delta S_1 \rightarrow \sigma^1 = g(\Sigma_{j=1}^1 \Delta S_j, \sigma^0) \]
\[ \Delta S_2 \rightarrow \sigma^2 = g(\Sigma_{j=1}^2 \Delta S_j, \sigma^1) \]
\[ \Delta S_3 \rightarrow \sigma^3 = g(\Sigma_{j=1}^3 \Delta S_j, \sigma^2) \]
\[ \Delta S_n \rightarrow \sigma^n \]
\[ \sigma_{it} = f^*(S_{itj}, \sigma^3) \]
Two Trimming Methods

- **Method 1 - Tagging + Resetting**
  - Identifies the set of vertices possibly affected by an edge deletion
  - These vertices are resetted back to the initial value
  - Guarantees safety with conservative trimming - however, slow

- **Method 2 - Active Value Dependence Tracking**
  - Tracks dynamic dependencies among vertices online
  - Leads to a much smaller set of affected vertices
  - The vertices are reset to a closer (safe) approximation instead of the initial value
Method 1 - Tagging + Resetting

- Upon a deletion, the target vertex of the deleted edge is tagged using a set bit.
- This tag is iteratively propagated - when an edge is processed where the source is tagged, the target is also tagged.
- To reduce the number of tagged vertices, rely on algorithmic insight - tag a vertex only if any of its in-neighbors that actually contributes to its current value is tagged.
  - In a typical monotonic algorithm, the value of a vertex is typically only computed from a single incoming edge.
- “Passive” technique - tagging is only performed upon edge deletion.
Method 1 - Tagging + Resetting

- Tagging conservatively tags every single vertex
- Tagging only the edges that actually contribute no longer tags A and C

(a) A simple graph.

(b) Computation using intermediate values after the deletion of A→D; shown in blue are the correct result.
Method 2 - Active Value Dependence Tracking

- Define a transitive, non-cyclic relation $\rightarrow$ that captures value dependencies
  - Say that $u \rightarrow v$ if there is an edge from $u$ to $v$ and $u$ actually contributes to $v$
  - Transitive: if $u \rightarrow v$ and $v \rightarrow w$, then $u \rightarrow w$
  - Non-cyclic: if $u \rightarrow v$, then $v$ does NOT $\rightarrow u$

- Why non-cyclic? Need to guarantee safety - if we need a safe approximate value for $v$, we can’t rely on any neighbor that was dependent on a past value of $v$

- This contributes-to relation needs to be defined by the developer (simple in practice)
Method 2 - Active Value Dependence Tracking

- Create a dependence tree
  - Acyclic
  - Every vertex has at most one incoming edge
- Non-accumulative - if a new value for a vertex is computed, that dependence replaces the old one

Now, KickStarter needs to compute new approximate values for the vertices affected by the deletion.
Method 2 - Active Value Dependence Tracking

1) Identify the set of vertices affected
   a) This is the subtree rooted at the target vertex of the deleted edge
   b) Ignore all other vertices

2) For each affected vertex $v$, compute a safe alternative value
   a) Resetting to the initial value also OK but slower
   b) Re-execute the update function on $v$ - however, CANNOT use any edge that was reliant on past values of $v$ (i.e. any vertex lower than $v$ in the dependence tree)
   c) How to quickly determine this? Use the level (depth) information in the dependence tree. Only consider edges that come from vertices whose level is $\leq$ level of $v$

3) Keep trimming if this safe alternative value disrupts monotonicity
   a) What direction is the monotonicity? Depends on the algorithm - in SSWP, values are monotonically increasing, in SSSP they are monotonically decreasing
   b) If the alternative value disrupts monotonicity, there might be something wrong with the children - have to trim those too
Method 2 - Active Value Dependence Tracking

(b) There are no safe incoming edges into D - thus, new value is 0. Monotonicity is disrupted.

(c) E gets incoming edges from C and D. B gets incoming edges from A and D

(d) Trimming continues to G
Method 2 - Active Value Dependence Tracking

- Good for performance too
- Look at previous SSSP problem - after edge is deleted, B’s value is immediately set to MAX (and C soon follows)
- Easy to parallelize too - computations are confined to a vertex and its neighbors

(a) A simple graph.  (b) Computation using the approximation after the deletion of A→B.
Experimental Results

- TAG = tagging + resetting
- VAD = value dependence trimming
- TOR = Tornado. Is not always correct.
- RST = no trimming, only resetting. Is always correct.
- 10% of edge updates are deletions

<table>
<thead>
<tr>
<th>Graphs</th>
<th>#Edges</th>
<th>#Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster (FT) [13]</td>
<td>2.5B</td>
<td>68.3M</td>
</tr>
<tr>
<td>Twitter (TT) [6]</td>
<td>2.0B</td>
<td>52.6M</td>
</tr>
<tr>
<td>Twitter (TTW) [20]</td>
<td>1.5B</td>
<td>41.7M</td>
</tr>
<tr>
<td>UKDomain (UK) [5]</td>
<td>1.0B</td>
<td>39.5M</td>
</tr>
<tr>
<td>LiveJournal (LJ) [3]</td>
<td>69M</td>
<td>4.8M</td>
</tr>
</tbody>
</table>

Table 2: Real world input graphs.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Issue</th>
<th>VERTEX_FUNCTION</th>
<th>SHOULD_PROPAGATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSWP</td>
<td>Correctness</td>
<td>$v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.\text{source.path}, e.weight))$</td>
<td>newValue &lt; oldValue</td>
</tr>
<tr>
<td>CC</td>
<td>Correctness</td>
<td>$v.component \leftarrow \min(v.component, \min_{e \in \text{edges}(v)} (e.\text{other.component}))$</td>
<td>newValue &gt; oldValue</td>
</tr>
<tr>
<td>BFS</td>
<td>Performance</td>
<td>$v.dist \leftarrow \min_{e \in \text{inEdges}(v)} (e.\text{source.dist} + 1)$</td>
<td>newValue &gt; oldValue</td>
</tr>
<tr>
<td>SSSP</td>
<td>Performance</td>
<td>$v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.weight + e.\text{source.path})$</td>
<td>newValue &gt; oldValue</td>
</tr>
</tbody>
</table>

Table 3: Various vertex-centric graph algorithms.
## Trimming for Correctness

<table>
<thead>
<tr>
<th></th>
<th>LJ</th>
<th>UK</th>
<th>TTW</th>
<th>TT</th>
<th>FT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSWP</strong></td>
<td><strong>RST</strong></td>
<td>7.48-10.16 (8.59)</td>
<td>81.22-112.01 (90.75)</td>
<td>94.18-102.27 (99.28)</td>
<td>170.76-183.11 (176.87)</td>
</tr>
<tr>
<td></td>
<td><strong>TAG</strong></td>
<td>11.57-14.71 (13.00)</td>
<td>1.73-62.1 (21.42)</td>
<td>27.38-125.91 (71.26)</td>
<td>262.88-278.42 (270.29)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>3.51-5.5 (4.48)</td>
<td>1.17-1.18 (1.17)</td>
<td>21.54-34.38 (27.55)</td>
<td>66.85-130.84 (75.88)</td>
</tr>
<tr>
<td><strong>CC</strong></td>
<td><strong>RST</strong></td>
<td>6.43-7.93 (7.19)</td>
<td>133.92-166.33 (148.80)</td>
<td>105.16-111.46 (107.54)</td>
<td>113.92-126.35 (126.35)</td>
</tr>
<tr>
<td></td>
<td><strong>TAG</strong></td>
<td>10.98-12.81 (11.86)</td>
<td>170.91-203.54 (183.93)</td>
<td>176.91-201.12 (185.84)</td>
<td>193.77-249.93 (208.90)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>4.89-5.85 (5.30)</td>
<td>1.81-7.75 (4.37)</td>
<td>31.78-33.24 (32.54)</td>
<td>21.98-22.58 (22.29)</td>
</tr>
</tbody>
</table>

Table 4: Trimming for correctness: query processing time (in sec) for SSWP and CC, shown in the form of min-max (average).

<table>
<thead>
<tr>
<th></th>
<th>LJ</th>
<th>UK</th>
<th>TTW</th>
<th>TT</th>
<th>FT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSWP</strong></td>
<td><strong>TAG</strong></td>
<td>3.1M-3.2M (3.1M)</td>
<td>8.9K-9.7M (4.1M)</td>
<td>1.1K-29.5M (13.4M)</td>
<td>28.5M-28.6M (28.6M)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>20.1K-90.8K (60.8K)</td>
<td>2.9K-93.0K (33.4K)</td>
<td>1.0K-4.5K (2.3K)</td>
<td>2.4K-1.1M (106.4K)</td>
</tr>
<tr>
<td><strong>CC</strong></td>
<td><strong>TAG</strong></td>
<td>3.2M-3.2M (3.2M)</td>
<td>25.9M-25.9M (25.9M)</td>
<td>31.3M-31.3M (31.3M)</td>
<td>32.2M-32.2M (32.2M)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>1.1K-3.1K (1.9K)</td>
<td>320.16K (1.0K)</td>
<td>116.463 (212)</td>
<td>241.463 (344)</td>
</tr>
</tbody>
</table>

Table 5: Trimming for correctness: # reset vertices for SSWP and CC (the lower the better) in the form of min-max (average).
Trimming for Correctness

Figure 8: Time taken to answer queries.
# Trimming for Performance

<table>
<thead>
<tr>
<th></th>
<th>LJ</th>
<th>UK</th>
<th>TTW</th>
<th>TT</th>
<th>FT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSSP</strong></td>
<td><strong>TOR</strong></td>
<td>1.27-102.88(39.10)</td>
<td>2.84-119.03(24.90)</td>
<td>17.62-131.9(112.57)</td>
<td>42.13-584.64(190.78)</td>
</tr>
<tr>
<td></td>
<td><strong>TAG</strong></td>
<td>3.25-4.49(3.97)</td>
<td>2.03-2.94(2.19)</td>
<td>46.06-52.5(48.96)</td>
<td>98.59-118.23(105.73)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>2.12-3.22(2.55)</td>
<td>1.33-1.5(1.41)</td>
<td>28.68-32.33(30.21)</td>
<td>41.35-48.65(44.19)</td>
</tr>
<tr>
<td><strong>BFS</strong></td>
<td><strong>TOR</strong></td>
<td>1.17-77.05(7.17)</td>
<td>1.24-588.09(142.55)</td>
<td>23.94-1015.76(199.23)</td>
<td>55-283.71(120.45)</td>
</tr>
<tr>
<td></td>
<td><strong>TAG</strong></td>
<td>3.47-4.43(3.88)</td>
<td>1.81-5.14(1.97)</td>
<td>51.08-58.3(54.36)</td>
<td>110.75-192.71(127.54)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>1.96-3.37(2.59)</td>
<td>1.21-3.88(1.42)</td>
<td>32.02-34.86(32.96)</td>
<td>69.43-91.88(74.27)</td>
</tr>
</tbody>
</table>

Table 6: Trimming for performance: query processing times (in sec) for SSSP and BFS in the form: min-max (average).

<table>
<thead>
<tr>
<th></th>
<th>LJ</th>
<th>UK</th>
<th>TTW</th>
<th>TT</th>
<th>FT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSSP</strong></td>
<td><strong>TAG</strong></td>
<td>8.2K-59.8K (25.9K)</td>
<td>4.1K-193.4K (36.4K)</td>
<td>19.7K-183.7K (89.4K)</td>
<td>6.2K-196.7K (51.5K)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>1.7K-40.1K (7.0K)</td>
<td>2.9K-52.2K (16.6K)</td>
<td>2.1K-77.7K (19.6K)</td>
<td>836-110.9K (11.1K)</td>
</tr>
<tr>
<td><strong>BFS</strong></td>
<td><strong>TAG</strong></td>
<td>10.8K-354.5K (79.0K)</td>
<td>1.3K-483.0K (35.5K)</td>
<td>20.9K-1.2M (457.6K)</td>
<td>44.2K-8.6M (1.1M)</td>
</tr>
<tr>
<td></td>
<td><strong>VAD</strong></td>
<td>5.5K-116.6K (36.4K)</td>
<td>3.2K-469.9K (41.2K)</td>
<td>860-3.1K (1.6K)</td>
<td>742-1.4K (1.1K)</td>
</tr>
</tbody>
</table>

Table 7: Trimming for performance: number of reset vertices for SSSP and BFS in the form: min-max (average).
Trimming for Performance

Figure 9: Trimming for performance: time taken to compute answer queries by TAG and VAD.
Experimental Results

- Kickstarter always produces correct results
- Much faster - speedup of 8.5 - 23.7
- Computing new approximate values in VAD drastically reduces # of reset vertices
- TAG is typically slower than VAD because it resets more vertices
- Dependence tracking overhead is only 13%