Journal of Big Data

RESEARCH Open Access

VeilGraph: incremental graph stream processing

Miguel E. Coimbra[*†], Sérgio Esteves[†], Alexandre P. Francisco[†] and Luís Veiga[†]

*Correspondence: miguel.e.coimbra@tecnico.ulisboa.pt
†Miguel E. Coimbra, Sérgio Esteves, Alexandre P. Francisco and Luís Veiga contributed equally to this work
INESC-ID/IST, Universidade de Lisboa, Rua Alves Redol 9, Lisboa 1000-029, Portugal

Abstract
Graphs are found in a plethora of domains, including online social networks, the World Wide Web and the study of epidemics, to name a few. With the advent of greater volumes of information and the need for continuously updated results under temporal constraints, it is necessary to explore alternative approaches that further enable performance improvements. In the scope of stream processing over graphs, we research the trade-offs between result accuracy and the speedup of approximate computation techniques. The relationships between the frequency of graph algorithm execution, the update rate and the type of update play an important role in applying these techniques. Herein we present VeilGraph, through which we conducted our research. We showcase an innovative model for approximate graph processing implemented in `Apache Flink`. We analyse the feasibility of our model and evaluate it with the case study of the PageRank algorithm, the most famous measure of vertex centrality used to rank websites in search engine results. Our experiments show that VeilGraph can often reduce latency closely to half (speedup of 2.0×), while achieving result quality above 95% when compared to results of the traditional version of PageRank executing in `Apache Flink` with `Gelly` (i.e. without any summarization or approximation techniques). In some cases, depending on the workload, speedups against `Apache Flink` reach up to 3.0x (i.e. yielding a reduction of up to 66% in latency). We have found VeilGraph implementation on Flink to be scalable, as it is able to improve performance up to 10X speedups, when more resources are employed (16 workers), achieving better speedups with scale for larger graphs, which are the most relevant.

Keywords: Graph processing, Approximate processing, Stream processing, Summarization, Dataflow programming, Distributed computation

Introduction

Graph-based data is increasingly relevant in big-data storage and processing, as it allows richer semantics in data and knowledge representation, as well as enabling the application of powerful algorithms with graph processing [1]. The domains of application are vast and varied with graphs being used to represent and process the structure and contents of the World Wide Web [2, 3], structures in chemistry [4], knowledge graphs [5], social networks [6], machine learning [7], analytics [8], epidemiology [9], transportation [10] and network security [11], to name a few.

Processing larger and larger graphs in a time and cost-effective way, even when static/immutable, requires distributed computing infrastructures, and distributed data processing platforms such as `Hadoop`, `Spark` and `Flink` have been adapted or extended to carry out graph processing (e.g., `GraphX/Spark` and `Gelly/Flink`).

With graphs being subject to dynamism, with updates arriving periodically or continuously (i.e., the latter called graph streaming), the challenge faced by these platforms is increased as the continuously arriving updates to the graph may require, without better knowledge, the entire graph to be (re-)processed by some algorithm in order to provide updated results to users' and applications' queries to the graph. This entails that with despite the larger volumes of information, there is also the need for continuously updated results under temporal constraints.

In this paper, we address the need to explore novel approaches that further enable performance improvements. In the scope of stream processing over graphs, we research the trade-offs between result accuracy and the speedup of approximate computation techniques. The relationships between the frequency of graph algorithm execution, the update rate and the type of update play an important role in applying these techniques.

We introduce VeilGraph, a novel execution model that enables approximate computations on general directed graph applications. Our model uses a summarized graph representation which includes only the vertices most relevant to computation using a set of heuristics over the topological changes in the graph. With this abstraction, we build a representative graph summarization that solely comprises the subset of vertices estimated as yielding a relevant impact to the accuracy of a given graph algorithm. This way, VeilGraph is capable of delivering lower latencies in a resource-efficient manner, while maintaining query result accuracy within acceptable limits.

We integrated VeilGraph with `Apache Flink` [12], a modern distributed dataflow processing framework. Experimental results indicate that our approximate computing model can achieve half the latency of the base (exact) computing model, while not degrading result accuracy by more than 5% (this was observed by comparing the complete and the approximate models with the same number of workers in the cluster). Furthermore, our approximate (summarized) computation model is scalable, achieving speedups in the range of 10x–15x while using only 16 workers in our Google Cloud Dataproc cluster experiments (we evaluated with 1, 2, 4, 8 and 16 worker counts in the cluster).

We focus on a vertex-centric implementation of PageRank [13], where for each iteration, each vertex u sends its value (divided by its outgoing degree) through each of its outgoing edges. A vertex v defines its score as the sum of values received from its incoming edges, multiplied by a constant factor β and then summed with a constant value $(1-\beta)$ with $0 \leq \beta \leq 1$. PageRank, based on the random surfer model, uses β as a dampening factor. For our work, this means that whether one considers one-time offline processing or online processing over a stream of graph updates, the underlying computation of PageRank is an approximate numerical version well known in the literature. This distinction is important, for when we state VeilGraph enables approximate computing, we are also considering a potential for applicability to a scope of graph algorithms, such as algorithms for computing centrality [14–16], heat kernel [17] and optimization algorithms for finding communities [18–20]. Whether the specific graph algorithm itself incurs numerical approximations (such as the *power method*) or not, that is orthogonal to our model and may only enable its benefits further.

This document is organized as follows. "Model: Big Vertex" section describes our summarization model and how it is built. An overview of the VeilGraph architecture is provided in "Architecture" section. In "Evaluation" section we present the experimental evaluation, followed by an analysis of improvements. "Related work" section addresses related systems and techniques. We summarize our contribution and identify future research in "Conclusion" section.

Model: Big Vertex

When a window of graph updates (e.g. a batch of edge additions and deletions) is incorporated into the graph, vertices change in different ways. The importance of a vertex with 5000 neighbours will typically not change much if 5 new vertices connect to it. But if the vertex only had 5 neighbours, it now has 10, it is a 100% increase and so the magnitude of its individual topological change is greater.

Our model considers a set K of *hot* vertices upon which computation of an algorithm (e.g. PageRank) will be performed, but only on those, contrary to executing computation on the entire set of vertices in the graph when results are queried/retrieved, as it is usual. The model resorts to a synthesis, unifying techniques such as defining and determining a confidence threshold for error in the calculation [21, 22], graph sampling [23–25] and sketching [26]. The aim of this set is to reduce the number of processed vertices as close as possible to $O(K)$, thus aiming to enable reductions in latency, and contributing to keeping latency within a given bound, irrespective of graph size. This can be specially relevant in cloud settings where latency-driven percentile SLAs are increasingly used (we refer to some previous work when we analyse and discuss the performance results).

In the model proposed in VeilGraph, for a given graph $G = (V, E)$, we build set K using three parameters (r, n, Δ). The vertices outside this set K assume the values they had (ranks) on the previous computation and are not updated in the current one. New vertices which were just obtained from the update stream are immediately added to K as they have no algorithm-specific (e.g. PageRank) value yet. Performing updates to only a subset K of the vertices implies that less data is propagated across the graph. In this model there is an aggregating vertex $\mathcal{B}$. We refer to $\mathcal{B}$ as the *big vertex*—a single vertex representing all the vertices not contained in K (in this model, the values are not updated for vertices in $\mathcal{B}$, thus saving computation effort and time).

For the original graph $G = (V, E)$, upon the arrival of edge additions/deletions, we build vertex set K and then define a summary graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = K \cup \{\mathcal{B}\}$. We define $\mathcal{E} = E_K \cup E_{\mathcal{B}}$, where $E_K = \{(u, v) \in E : u, v \in K\}$, which is the set of edges with both source and target vertices contained in K and $E_{\mathcal{B}} = \{(w, z) \in E : w \notin K, z \in K\}$ as the set of edges with sources contained *inside* $\mathcal{B}$ and target in K.

Conceptually, this consists in replacing all vertices of G which are not in K by a single big vertex $\mathcal{B}$ and representing the edges whose targets are *hot vertices* and whose sources are now in $\mathcal{B}$. The summary graph $\mathcal{G}$ does not contain vertices outside of K (again, those are represented by $\mathcal{B}$). By definition, $\mathcal{B}$ represents all vertices whose impact is not expected to change significantly. The contribution of each vertex $v \notin K$ (and therefore represented by $\mathcal{B}$) is constant between iterations (e.g., random walks), so it can be registered initially before updating algorithm values (e.g., rankings) and used afterwards during algorithm execution.

As a consequence, the summary graph $\mathcal{G}$ does not contain edges targeting vertices represented by $\mathcal{B}$. However, their existence must be recorded: even if the edges coming out of K and into $\mathcal{B}$ are irrelevant for the computation, they still matter for the vertex degree, which influences the emitted scores of the vertices in K. Despite the fact those edges targeting $\mathcal{B}$ are being discarded when building the summary graph $\mathcal{G}$, the summarized computation must occur as if vertex degrees remained the same. To ensure correctness, for each edge $(u, v) \in E_K$, we store and represent the weight of the edge as $w(u, v) = 1/d_{out}(u)$ with $d_{out}(u)$ as the out-degree of u before discarding the outgoing edges of u targeting vertices in $\mathcal{B}$.

It is also necessary to record the contribution of all the vertices fused together in $\mathcal{B}$. For each edge (u, v) whose source u is inside $\mathcal{B}$ and whose target v is in K, we store the contribution that would originally be sent from u as $w(u, v) = u_s/d_{out}(u)$ where u_s is the stored value of u (resulting from the computed graph algorithm) and the out-degree of u is defined as $d_{out}(u)$. The contribution of $\mathcal{B}$ as a single vertex in $\mathcal{G}$ is then represented as $\mathcal{B}_s$ and defined as:

$$\mathcal{B}_s = \sum_u w(u, v), (u, v) \in E_{\mathcal{B}}. \tag{1}$$

The fusion of vertices into $\mathcal{B}$ is performed while preserving the global influence from vertices placed inside $\mathcal{B}$ to vertices in K. Our model intuition is that vertices receiving more updates have a greater probability of having their measured impact change in between execution points. Their neighbouring vertices are also likely to incur changes, but as we consider vertices further and further away from K, contributions are likely to remain the same or close to unmodified [27, 28].

To build K after integrating a window of updates, the three parameters we consider are:

- *Parameter 1: Update ratio threshold r.* This parameter defines the minimal amount of change in vertex degree in order for it to be included in K. A vertex whose in-degree changes r% or more is included in K immediately (e.g., a vertex that changed in-degree from two to three changed by 50%).

  We adopt the notation where the set of neighbours of vertex u in a directed graph at measurement instant t is written as $N_t(u) = \{v \in V : (u, v) \in E_t\}$. We further write the degree of vertex u in measurement instant t as $d_t(u) = |N_t(u)|$. The function $d(u, v)$ represents the length (number of hops) of the minimum path between vertices u and v and $d_t(u, v)$ represents the same concept at measurement instant t. It is not required to maintain shortest paths between vertices (that would be a whole different problem [29]). This model is based on a vertex-centric breadth-first neighbourhood expansion. Let us define as K_r the set of vertices which satisfy parameter r, where $d_t(u)$ is the degree of vertex u, t represents the current measurement instant and $t - 1$ is the previous measurement instant:

$$K_r = \left\{ u : \left| \frac{d_t(u)}{d_{t-1}(u)} - 1 \right| > r \right\}. \tag{2}$$

New vertices are always included in K. The subtraction in the formula registers the degree change ratio with respect to the previous value $d_{t-1(u)}$. This definition allows us to mathematically express conditions such as *keeping all vertices whose degree changed at least 20%*.

- *Parameter 2: Neighbourhood diameter n.* We expand around the n-hop neighbourhoods of the vertices added to K in step 1. The ones found through this local expansion are also included in K.
  It aims to capture the locality of the impact in graph updates: those vertices neighbouring the ones beyond the threshold, and as such still likely to suffer relevant modifications when (the rank of) vertices in K are recalculated (attenuating as distance increases). On measurement instant t, for each vertex $u \in K_r$, we will expand a neighbourhood of diameter n, starting from u and including every additional vertex $v \in V_t \backslash K_r$ found in the neighbourhood diameter expansion. The expansion is then defined as:

$$K_n = \{v : d_t(u,v) \leq n, u \in K_r, v \in V_t \setminus K_r\}. \tag{3}$$

  V_t is the set of vertices of the graph at measurement instant t. $n = 0$ may be set to promote performance, while a greater value of n is expected to focus on accuracy at the expense of performance.
- *Parameter 3: Result-specific neighbourhood extension* Δ. Between computations, the values change. A vertex u whose value changed at least Δ between computations may have a greater (this being application-specific)[1] influence on its neighbours until it becomes negligible after a given number of hops. We select these influenced neighbours based on the formula:

$$K = K \cup \{v : d(u,v) \leq f_\Delta(v), \qquad u \in K, v \in V \setminus \{K\}\}, \tag{4}$$

  where $f_\Delta(v)$ is the Δ-expansion function:

$$f_\Delta(v) = \frac{1}{\log \overline{d}} \log\left(\frac{\overline{d}\, v_s}{\Delta\, d(v)}\right). \tag{5}$$

  Remaining symbols: $\overline{d}$ is the average degree in the graph, v_s is the existing result on vertex v and $d(v)$ is the out-degree of v. The intuition underlying this parameter is the following: for a vertex u that will be expanded via Δ from step 3, its impact will diminish as we hop further away from it (from u to immediate neighbours, then to its neighbours' neighbours and so on). The impact of a vertex u on its i-hop neighbourhood will dilute as we further hop away from u as $i \to \infty$. Consider we perform a number i of hops away from u until we reach a given vertex v. Δ is used with this

[1] This is orthogonal to changes in degree, e.g., while degree may stay constant, the inflow of rank received by a vertex may change significantly due to the change of rank of one or more of its neighbouring or near vertices. Thus, the model is able to capture and propagate only significant changes—here, in rank—across the parts of the graph topology where they are deemed relevant, in an application-specific way.

formula to assign a value to the number of hops. With this, we can account for vertices which change in an application-specific impacting way, so that their neighbourhoods are included in the computation as well.

The vertices outside K are aggregated into a *big vertex* $\mathcal{B}$. We define $\mathcal{E} = E_K \cup E_{\mathcal{B}}$, where $E_K = \{(u, v) \in E : u, v \in K\}$ is the set of edges with both source and target vertices contained in K and $E_{\mathcal{B}} = \{(w, z) \in E : w \notin K, z \in K\}$ as the set of edges with sources contained *inside* $\mathcal{B}$ and target in K. To ensure correctness, for each edge $(u, v) \in E_K$, we store and represent the weight of the edge as $w(u, v) = 1/d(u)$ with $d(u)$ as the out-degree of u before discarding the outgoing edges of u targeting vertices in $\mathcal{B}$. This is so that the correct degree of the vertices in K is used in the computation of scores, even though their outgoing edges (with targets in $\mathcal{B}$) are not used in the summarization. Thus, the vertices represented in $\mathcal{B}$ are assumed to not change their value during the computation (which may lead to error accumulation – we analyse and deal with it in Sec. 4). What is captured and preserved is their global contribution to the vertices in K. We then have a summary graph written as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = K \cup \{\mathcal{B}\}$.

Equation 5 is based on the diminishing returns incurred by continuously expanding the neighbourhood when building K_Δ. When the result of a vertex v (e.g., its PageRank) is propagated to a 1st-order neighbour u of v, the score gets diluted by the amount of out-edges of v. So if v has a result of v_s, each of its immediate neighbouring vertices will receive $v_s/d_t(v)$ at measurement instant t. If we were to further propagate to 2nd-hop neighbours of v, we would expand from vertex u into a farther vertex w. The value received by w would be $v_s/(d_t(v)d_t(u))$. Expanding into an ith-hop neighbour would accumulate an out-degree division for each vertex expanded.

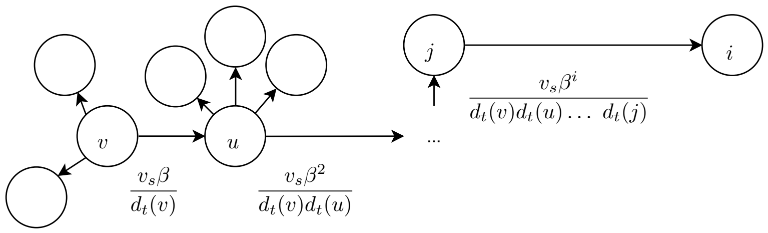
The score contribution is also dampened by the impact of β (known as the dampening factor in PageRank) with each hop, which we represent in Eqs. 6 and 7. The impact of v's result on a given i-th-hop vertex i would be defined as seen below (x is the last expanded vertex).

$$\frac{v_s \beta^i}{d_t(v) d_t(u) \ldots\ d_t(x)}. \tag{6}$$

To avoid fetching the out-degree of each vertex for each expansion (which would incur additional overhead in a dataflow processing system such as `Flink`), we approximate the degree of any vertex beyond v with $\overline{d}$, which is the average degree of the accumulated vertices with respect to the stream. We then have the approximation:

$$\frac{v_s \beta^i}{d_t(v) d_t(u) \ldots\ d_t(x)} \approx \frac{v_s \beta^i}{d_t(v) (\overline{d})^{i-1}}. \tag{7}$$

To ensure that we only continue expanding until we drop below a given Δ threshold of result impact (until the ith-hop vertex), we set it as a target of Eq. 7:



v
u
j
i
...
$\frac{v_s\beta}{d_t(v)}$
$\frac{v_s\beta^2}{d_t(v)d_t(u)}$
$\frac{v_s\beta^i}{d_t(v)d_t(u)\ldots\ d_t(j)}$

Fig. 1 The contribution of vertex *v* diminishes as we expand further away from it (Eq. 7). Parameter Δ dictates how far to expand around vertices until the accumulated fraction drops below Δ. v_s is the score of vertex *v*; $d_t(v)$ is the out-degree of vertex *v* at measurement instant *t*

$$\begin{aligned}
\frac{v_s\beta^i}{d_t(v)(\overline{d})^{i-1}} &= \frac{\overline{d}v_s\beta^i}{d_t(v)(\overline{d})^i} = \Delta \\
\frac{\overline{d}v_s}{\Delta\ d_t(v)} &= \left(\frac{\overline{d}}{\beta}\right)^i \\
\log_{\overline{d}/\beta}\left(\frac{\overline{d}v_s}{\Delta\ d_t(v)}\right) &= i \\
\frac{1}{\log\overline{d}/\beta}\log\left(\frac{\overline{d}v_s}{\Delta\ d_t(v)}\right) &= f_\Delta(v) = i.
\end{aligned}$$

Figure 1 provides a visual example of the dilution of the score of vertex v due to the degree of the vertex at the end of each hop (u, then intermediate vertices represented as '...', followed by j until i is reached).

We then have a set of *hot vertices* $K = K_r \cup K_n \cup K_\Delta$ which is used as part of a graph summary model (deriving from techniques in iterative aggregation [30]), written as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

An example of applying the updates and building the K set and the big vertex $\mathcal{B}$ is shown in Fig. 2 with $r = 0.20, n = 1, \Delta = 0.10$: *a*) is the initial graph; *b*) shows five new edges (dashed) and one edge deletion (red cross); *c*) new vertices are automatically included in K for computation (blue); *d*) vertices 0 and 1 are also included in K (blue) because their in-degree changed at least 20% ($r = 0.20$); *e*) the neighbourhood diameter expansion of size $n = 1$ around current vertices in K includes vertex 2; *f*) vertices (3, 4, 5, 6) outside K are collapsed into the *big vertex* $\mathcal{B}$. Impact of Δ not depicted.

The approximation model of VeilGraph is such that we want to prioritize intense and localized graph updates (e.g., new vertices or those whose number of neighbours shifts by a considerable amount due to parameter r. In that hypothetical scenario, where the majority of vertices has more than 500 neighbours, the impact of these new vertices will be greater locally on the vertex that doubled its number of neighbours. For scalability, we do not rely on any global information about the graph. That kind of reasoning can be incorporated through the Δ parameter. However, by virtue of being application-specific, this may require additional knowledge and insight of the specific application (i.e. processing algorithm) semantics and dynamics, in order to fine-tune.

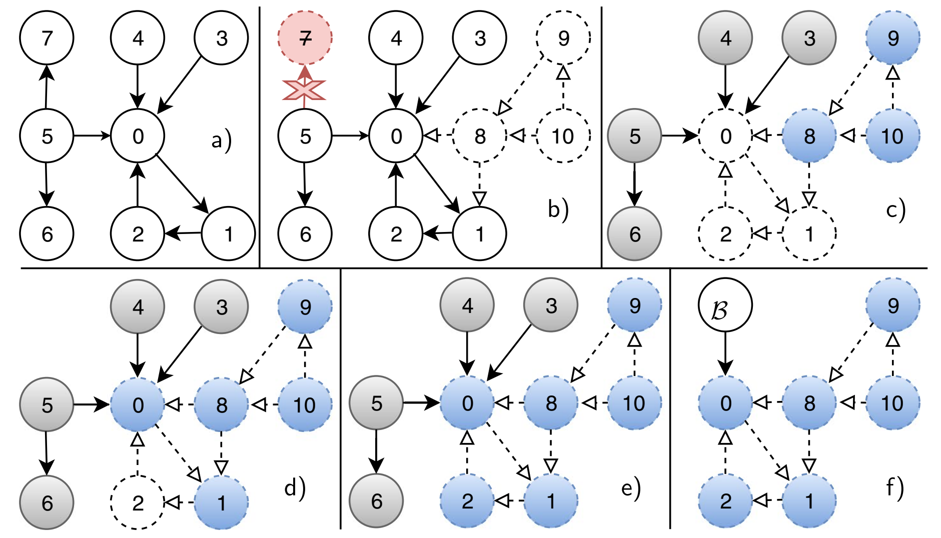


Fig. 2 Building the big vertex, step-by-step

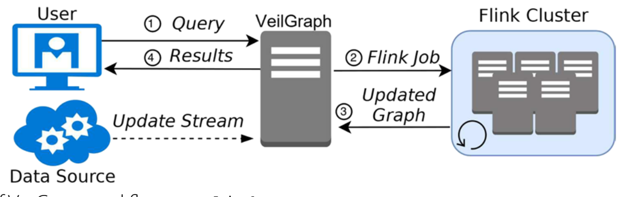


Fig. 3 Diagram of VeilGraph workflow on `Flink`

Architecture

The general distributed architecture and workflow of VeilGraph is illustrated in Fig. 3. The architecture of VeilGraph was designed to take into account the relevant stages in the graph processing as streaming updates arrive. The VeilGraph module is primarily responsible for continuously monitoring one or more streams of data and track the updates to be applied to the graph. When a graph algorithm is to be run (we henceforth call this occurrence a *query*—Step 1), VeilGraph will execute the request by submitting a job to the `Flink` cluster (Step 2). The view of VeilGraph over the graph will be updated (Step 3) and the results, possibly resulting from an approximate processing of the graph, are returned to the user (Step 4).

In our experiments, we trigger the incorporation of updates into the graph whenever a client query arrives.[2] To simplify implementation design, the client queries are also sent in the stream of graph updates. Conceptually, these are the major elements involved in the functioning of VeilGraph and are compatible with most graph processing frameworks:

- *Initial graph G*. The original graph upon which updates and queries will be performed.

[2] While outside the scope of this work, a live scenario would have a more elaborate ingestion scheme, possibly using dedicated ingestion nodes like in `KineoGraph` [40].

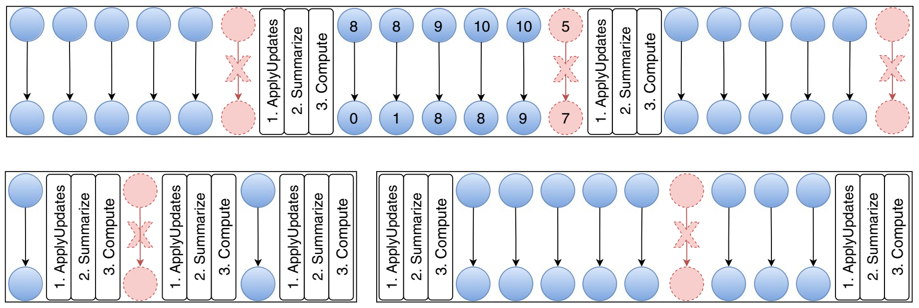


Fig. 4 Example of different edge addition/removal granularities in streams

- *Stream of updates S*. Our model of updates could be the removal e_- or addition e_+ of edges and the same for vertices (v_-, v_+). We make as little assumptions as possible regarding S: the data supplied needs not respect any defined order. In our experiments we used both edge additions and removals.
- *Result R*. Information produced by the system as an answer to the queries received in S (e.g. vertex rankings). It is reused in the following computation when the next window of updates and query are received.

Figure 4 illustrates different examples of update streams. In our stream scenario, these are non-overlapping counting windows. In the top stream, we see that each window of updates adds five edges and removes one. The greater the amount of elements in the stream window, the more entropy each update would potentially add to the graph. The bottom-right shows a bigger window and the bottom-left shows the extreme case where the element count in the window is one. The smaller the window size, the more resources are consumed (by recalculating ranks completely more often), and the greater the computational benefit, increased speedup and reduced latency of applying our model, as there are practically no changes to the graph.

VeilGraph was designed to allow programmers to define fine-grained logic for the approximate computation when necessary. This is achieved through user-defined functions defined in a specific base `Java` class `GraphStreamHandler`. The API of VeilGraph uses them to define the execution logic that will guide the processing strategy. They are key points in execution where important decisions should take place (e.g., how to apply updates, how to perform monitoring tasks). To employ our module, the user can express the algorithm using `Flink` dataflow programming primitives. To implement other graph algorithms, users can simply extend the VeilGraph `Java` class implementing the logic shown in Algorithm 1 to enable our model's functionality, while abstracting away many implementation details (inherent to our architecture) unrelated to the graph processing itself.

Additional behaviour control is possible by customizing the model by implementing user-defined functions (left as `abstract` methods of the class implementing the architecture logic). Overall, this approach has the advantage of abstracting away the API's complexity, while still empowering users who wish to create fine-tuned policies. VeilGraph 's architecture creates a separation between the graph model, the way the graph processing is expressed (e.g. such as vertex-centric) and the function logic to apply on vertices. We employ `Flink's` mechanism for efficient dataflow iterations [31] with intermediate result usage, expressing computations over its `Gelly` graph library.[3]

```
Algorithm 1 VeilGraph Execution Skeleton:  graph G, stream S
1: OnStart(G, S)   /* Initializations. */
2: graphUpdates ← ∅
3: updateStatistics ← ∅
4: repeat
5:    msg ← TakeMessage(S)
6:    if msg is Add then RegisterAddEdge(msg, graphUpdates, updateStatistics)
7:    else if msg is Remove then RegisterRemoveEdge(msg, graphUpdates,
   updateStatistics)
8:    else if msg is Query then
9:        needToApplyUpdates? ← CheckUpdateState(graphUpdates, updateStatistics)
10:       if needToApplyUpdates? then
11:          G ← ApplyUpdates(graphUpdates, updateStatistics)
12:       end if
13:       strategy ← DefineQueryStrategy(msg, G, updates, updateStatistics)
14:       if strategy = Repeat-last-answer then
15:          newResults ← previousResults
16:       else if strategy = Compute-approximate then
17:          newResults ← ComputeApproximate(G, previousResults)
18:       else if strategy = Compute-exact then
19:          newResults ← ComputeExact(G)
20:       end if
21:       OutputResults(newResults)
22:       OnQueryResult(msg, G, newResults, jobStatistics)  /* Extrapolate and store
   job statistics. */
23:    end if
24: until stopped
25: OnStop( )  /* Tear-down procedure. */
```

We present in Alg. 1 these different User-Defined Functions (UDFs) and their coordination. The functions are as follows: For simple rules, these functions don't need to be programmed, as we supply the implementation with parameters for the simplest rules such as threshold comparisons, fixed values, intervals and change ratios.

1 OnStart. A preparatory function for setting up resources such as files, database accesses or other initial tasks.
2 CheckUpdateState. Executed after a query q is received, but before graph updates are applied. Its purpose is to enable programmers to choose how and when to process the graph updates as a function of the magnitude of their impact. It exposes the sequence of graph operations which were pending since the last query and statistics such as the number of changed vertices and the total amount of vertices and edges in the graph.
3 DefineQueryStrategy. Called every time a query q arrives. The query is served after any processing that may have taken place in BeforeUpdates. This function is defined in the API to return an action indicator dictating the query strategy to use. It could be done be any of: (a) by returning the last calculated result; (b) performing an approximation of the result and returning it; (c) providing an exact answer after a complete recalculation of the result.[4]

[3] https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/iterations.html#delta-iterate-operator.

[4] In our scenario we always used (*b*)) approximation.

4 OnQueryResult. Invoked after q's response has been processed. This UDF is aware of the action indicator returned by DefineQueryStrategy. It has access to the response's results, execution statistics (such as total execution time, physical space, network traffic, among others) and details specific to the approximation technique used.
5 OnStop. Symmetrical to OnStop, it is responsible for (if necessary) proper resource clearing and post-processing.

Implementation

As already referred, VeilGraph was implemented on `Apache Flink` [12], a framework built for distributed stream processing.[5] It has many different libraries, among which `Gelly`, its official graph processing library. It features algorithms such as PageRank, Single-Source Shortest Paths and Community Detection, among others. Overall, it empowers researchers and engineers to express graph algorithms in familiar ways such as the *gather-sum-apply* or the *vertex-centric* approach of `Pregel` [32], while providing a powerful abstraction with respect to the underlying scheme of distributed computation. We employ `Flink's` mechanism for efficient dataflow iterations [31] with intermediate result usage. To employ our module, the user can express the algorithm using `Flink` dataflow programming primitives. They will be fed the updated graph and the processing infrastructure of `Flink`.

Evaluation

The source of VeilGraph is available online for the graph processing community.[6] We provide an API allowing programmers to implement their logic succinctly. VeilGraph was evaluated with the PageRank power method algorithm [33]. The PageRank logic is succinctly implemented as a function as follows:

```java
public static class PageRankFunction implements
    Function<MessageIterator<Double>, Double>, Serializable {

       private final Double dampening;

       public PageRankFunction(Double dampening) {

           this.dampening = dampening;
       }

       @Override
       public Double apply(final MessageIterator<Double> inMessages) {

           double rankSum = 0.0;
           for (double msg : inMessages) {
               rankSum += msg;
           }
           return (this.dampening * rankSum) + (1 - this.dampening);
       }
   }
```

[5] https://flink.apache.org/.

[6] Access date: 2021-Aug-26: https://github.com/mcoimbra/VeilGraph/.

This is then passed on to the underlying graph processing paradigm, as such:

```
PageRankFunction prf = new PageRankFunction(dampeningFactor);
GraphAlgorithm<Long, Double, Double, DataSet<Tuple2<Long, Double>>> algo =
    new VertexCentricAlgorithm(iterations, prf);
DataSet<Tuple2<Long, Double>> ranks = summaryGraph.run(algo);
```

While we focus our evaluation on PageRank, we note that other random walk based algorithms can be expressed easily. In our PageRank implementation, all vertices are initialized with the same value at the beginning.

Experimental setup To realistically evaluate the effect that cluster execution has on speedup, we evaluated our datasets in Google Cloud Dataproc[7] clusters with different worker counts (2, 4, 8, 16). Each machine was created with the `custom-4-26368` flag of the `gcloud` shell utility and runs an image based on `Debian 4.9.168-1+deb9u5`. We used `Flink 1.9.1` configured to use a parallelism of one within each worker. In our scenario, PageRank is initially computed over the complete graph G and then we process a stream S of windows of incoming edge updates. For each window received from the stream we: (1) integrate the edge updates into the graph; (2) compute the summarized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as described in "Model: Big Vertex" section and execute PageRank over $\mathcal{G}$. Thus, we *process* a query when PageRank (summarized or complete) is executed after integrating a window of updates.

Note that employing smaller windows, i.e., down to size one and towards continuous complete vertex rank updates, on every graph update arriving in the stream (even if without relevant changes in outcome expected, acting as a *strawman* competing alternative) would only further amplify VeilGraph 's benefits.

Stream *S* size and configuration

We do not use a window of updates with size of 1, as that would favour our model's summary graph $\mathcal{G}$ when compared against the repetitive complete execution over the whole graph G. To reduce variability, the stream S of edge updates was set up so the number Q of queries for each dataset and parameter combination is always fixed: fifty ($Q = 50$). Additionally, for each dataset, streams were generated by uniformly sampling from the edges in the original dataset file. A stream size of $|S| = 40000$ was used, implying $|S|/Q = 800$ edges are added before executing every query.

The number of stream queries Q and number of edge additions per query update were chosen to favour (non-approximate) `Flink` for comparability in a sensible and realistic scenario against summarized executions. For $|S| = 40,000$, if we added 8 edges instead of 800 before executing each query, we would have $Q = 5000$. This is a much longer sequence of queries, where the graph barely changes between them, with VeilGraph having near-zero execution times in most, where `Flink` would be 100-fold slower processing the complete graph. To avoid that, we empirically chose the value of 800 edges before each query (resulting in $Q = 50$).

[7] Access date: 2021-Aug-26: https://cloud.google.com/dataproc.

We test both edge additions and deletions. Every time we add edges, we remove an amount equal to 20% of the number of edges added. The edges to remove are chosen at random with equal probability. When additions and removals are applied before an execution, the removal only targets remaining edges which already existed in the original graph or that were added in an older update that preceded a previous execution: in any update window, we do not add and remove new edges, as that would have no effect.

For each dataset and stream *S*, each combination of parameters r, n, Δ is tested against a replay of the same stream. Essentially, each execution (representing a unique combination of parameters) will begin with a complete PageRank execution followed by $Q = 50$ summarized PageRank executions. This initial computation represents the real-world situation where the results have already been calculated for the whole graph previously. In such a situation, one is focused on the new incoming updates. For each dataset and stream *S*, we also execute a scenario which does not use the parameters: it starts likewise with a complete execution of PageRank, but the complete PageRank is always executed for all *Q* queries. This is required to obtain ground-truth results to measure accuracy and performance of the summary model.

Many datasets such as web graphs are usually provided in an incidence model [18, 34]. In this model, the out-edges of a vertex are provided together sequentially. This may lead to an unrealistically favourable scenario, as it is a property that will not necessarily hold in online graphs and which may benefit performance measurements. To account for this fact, we previously shuffle the stream *S*. A single shuffle was performed a priori for all datasets so that the randomized stream is the same for different parameter r, n, Δ combinations that were tested. This increases the entropy and allows us to validate our model under fewer assumptions, and assess it to hold in general scenarios.

Datasets

The datasets' vertex and edge counts are shown in Table 1. We evaluate results over two types of graphs: web graphs and social networks. They were obtained from the Laboratory for Web Algorithmics [34]. These datasets were used to evaluate the model against different types of real-world networks.

Assessment metrics

We measure the results of our approach in terms of the ability to delay computation in light of result accuracy; obtained execution speedup with increasing number of workers; reduction in number of processed edges. Accuracy in our case takes on special importance and requires additional attention to detail. The PageRank score itself is a measure of vertex/page importance and we wish to compare rankings obtained on a summarized execution against rankings obtained on the non-summarized graph. As such, what is desired is a method to compare rankings.

Rank comparison can incur different pitfalls. If we order the list of PageRank results in decreasing order, only a set of top-vertices is relevant. After a given index in the ranking, the centrality of the vertices is so low that they are not worth considering for comparative purposes. But where to define the truncation? The decision to truncate at a specific position of the rank is arbitrary and leads to the list being incomplete. Furthermore, the contention between ranking positions is not constant. Competition is much more

Table 1 Datasets: Laboratory for web algorithmics [34]

Dataset	|V|	|E|	#Pairs	$\overline{d}$ $max(d_{in})$ $max(d_{out})$	%Dang.	C_1	I_G
`cnr-2000`[1]	0.325 M	3.216 M	35.38% (± 0.329)	9.879 18,235 2716	23.98%	112,023 (34.41%)	17.45 (± 0.041)
`eu-2005`[1]	0.862 M	19 M	87.01% (± 0.789)	22.297 68,922 6985	8.31%	752,725 (87.26%)	10.18 (± 0.037)
`eu-2015-host`[1]	11 M	386 M	57.60% (± 0.119)	34.350 174,433 398,600	21.60%	6.512 M (57.82%)	5.83 (± 0.002)
`dblp-2010`[2]	0.326 M	1.615 M	47.74% (± 0.483)	4.952 238 238	7.83%	226,413 (69.41%)	7.35 (± 0.012)
`amazon-2008`[2]	0.735 M	5.158 M	84.40% (± 0.695)	7.015 1076 10	12.04%	627,646 (85.36%)	12.06 (± 0.021)
`holly-wood-2011`[2]	2.181 M	229 M	76.56% (± 0.757)	105.003 13,107 13,107	8.96%	1.917 M (87.91%)	3.926 (± 0.005)

Web graphs are indicated with [1] and social networks with [2]

intense between the first and second-ranked vertices than between the two-hundredth and two-hundredth and first.

To account for this in a sound manner, we employed Rank-Biased-Overlap (RBO) [35] as a meaningful evaluation metric (representing relative accuracy) developed to deal with these inherent issues of rank comparison. RBO has useful properties such as weighting higher ranks more heavily than lower ranks, which is a natural match for PageRank as a vertex centrality measure. It can also handle rankings of different lengths. This is in tune with the output of a centrality algorithm such as PageRank. The RBO value obtained from two rank lists is a scalar in the interval [0, 1]. It is zero if the lists are completely disjoint and one if they are completely equal. While more recent comparison metrics have been proposed [36], they go beyond the scope of what is required in our comparisons.

Performance-wise, we test values of r associated to different levels of sensitivity to vertex degree change (the higher the number, the less expected objects to process per query). With $n = 0$, we minimize the expansion around K and consider just the vertices that passed the degree change of r%. For $n = 1$, we are taking a more conservative approach regarding result accuracy. An overall tendency to expect is that the higher the value of n is, the higher the RBO. The Δ values were chosen to evaluate individual different weight schemes applied to vertex score changes. The relation between parameters r and n has a greater impact in performance and accuracy than the relation of any of these parameters with Δ. We tested with two sets of parameter combinations:

- RBO-oriented $(r = 0.05, n = 2, \Delta = 1.0)$, $(r = 0.05, n = 2, \Delta = 0.5)$, $(r = 0.05, n = 6, \Delta = 1.0)$, $(r = 0.05, n = 6, \Delta = 0.5)$. This has a very low threshold of sensitivity to the ratio of vertex degree change $(r = 0.05)$.

- Performance-oriented $(r = 0.20, n = 0, \Delta = 0.5)$, $(r = 0.20, n = 0, \Delta = 1.0)$, $(r = 0.20, n = 1, \Delta = 0.5)$, $(r = 0.20, n = 1, \Delta = 1.0)$, $(r = 0.20, n = 4, \Delta = 1.0)$. With $r = 0.20$, the goal is to be less sensitive pertaining degree change ratio.

For both of these combinations, we test with low and high values of n to examine how expanding the neighbourhood of vertices complements the initial degree change ratio filter. Using a higher number of ranks for the RBO evaluation favours a comparison of calculated ranks which has greater resolution, as more vertices are being compared. In our evaluation, the RBO of each execution is calculated using 10% of the complete graph's vertex count as the number of top ranks to compare. We address the aforementioned issue of truncation by making the number of truncated ranks specific to each dataset by defining it as a percentage of the graph's vertices. Considering the nature of the rankings, we focus on comparing the top 10% of vertex ranks of the complete and summarized execution scenarios using RBO, as it is in this top that the most relevant ones are concentrated. Furthermore, every 10 executions, we calculate RBO using all of the vertices of the graph to periodically ensure that no artefacts are masked in the lower rank values.

Results

Parameters (r, n, Δ) producing the best accuracy result are not necessarily, expectably so, the same ones producing the best speedups. The horizontal axis represents the same (except for the candle bar and regular bar plots) for all plots: it is the sequence of queries from 1 up to $Q = 50$. Due to how the dynamics of parameter combinations and the structure of the data sets behave, some parameter combinations produced highly similar values, leading to almost overlapping plots.

One needs to take into account this is a challenging assessment context for VeilGraph aimed to reflect a sensible and realistic scenario. In fact, between each consecutive pair of the 50 queries (i.e., on every of the 800 edge/vertex updates we are ingesting between them), if the user prompted a query execution, VeilGraph could offer near-instant results leveraging the previously summarized graph, and thus yielding several 100-fold speedups against a full graph execution (while true, that would be the *strawman* competing alternative). This, while still providing results with very high RBO (in line with those from the preceding and successor of the pair of queries where the update lied between). In accordance, speedup candle bar values were obtained by calculating the average and standard deviation values of the computational time across the $Q = 50$ executions.

Datasets and benefits of the summary model against the complete graph execution Figures 5, 6, 7 and 8 display for several datasets (for each of the $Q = 50$ executions in each test) the speedups representing the relation between the complete graph PageRank computation time and the VeilGraph summary model computation time. For each dataset, we present the best three and worst three parameter combinations in terms of resulting speedup.

`cnr-2000` speedup: the best speedup achieved was 1.20 for parameters $(r = 0.20, n = 0, \Delta = 1.00)$ as shown on Fig. 5 with the blue star markers. For these,

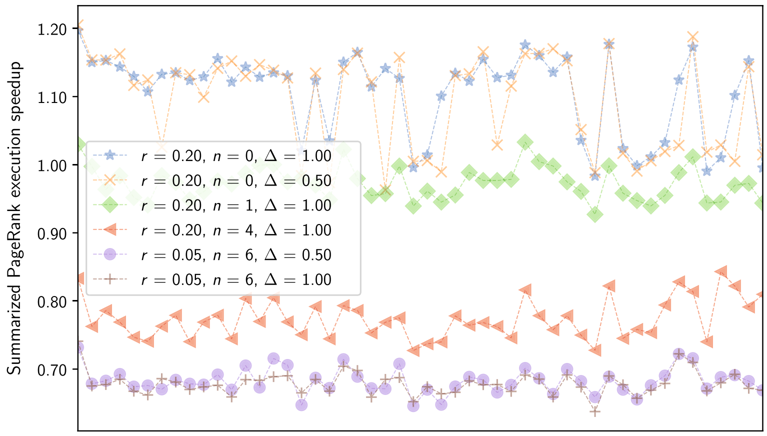


Fig. 5 `cnr-2000` speedups: summary model versus complete graph

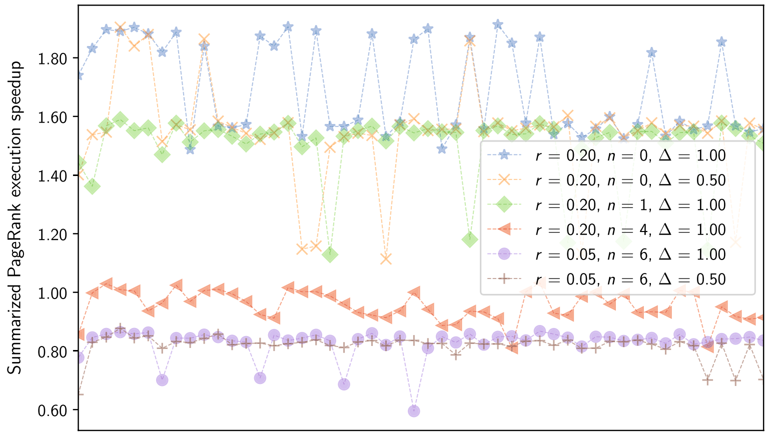


Fig. 6 `dblp-2010` speedups: summary model versus complete graph

increasing n from 0 to 1 produced an execution speed similar to the baseline (the execution time of complete executions). This is to be expected as the scope of computation increases due to a bigger *hot vertex* set K. The worst speedups were obtained with $(r = 0.05, n = 6)$, a combination which promotes accuracy above everything. These actually performed slower than the baseline complete executions due to the overhead of summarization model construction and computation. This dataset has an average distance of $l_G = 17.45$ and its largest component contains only $C_1 = 34.41\%$ of the vertices. This single component represents a small percentage of the graph compared to other datasets. Coupled with the high average distance, it limits the benefit of the big vertex model, although it still yields speedups. It also has a high number of dangling nodes (23.98%), increasing computational overhead. The most influential vertex in this dataset has $d_{out} = 2716$ and the most influenced one has $d_{in} = 18235$. We attribute cascading effects to these properties.

`dblp-2010` speedup: the best speedups obtained were around 1.60–1.80 for high values of the update ratio threshold $(r = 0.20)$ and lower levels of neighbourhood

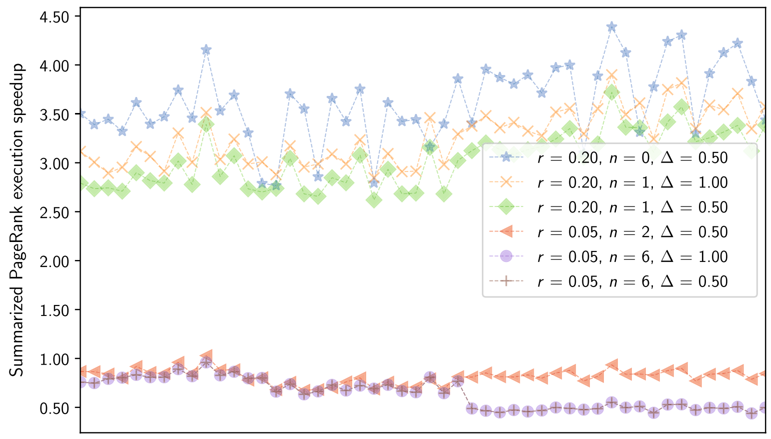


Fig. 7 `eu-2005` speedups: summary model versus complete graph

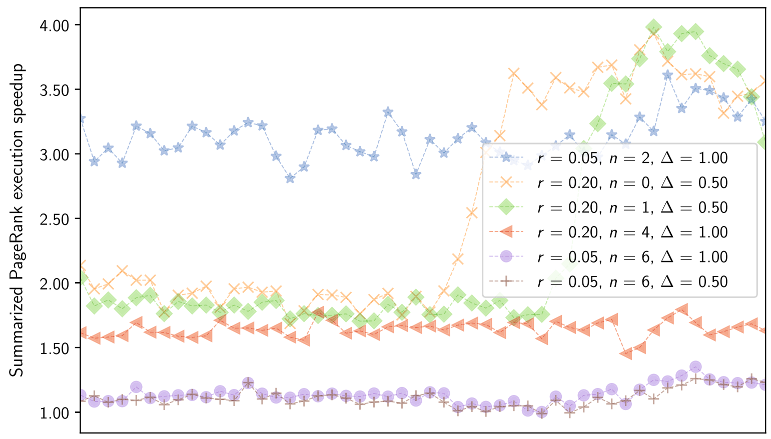


Fig. 8 `amazon-2008` speedups: summary model versus complete graph

expansion ($n = 0, n = 1$). These are the markers with blue stars, yellow crosses and green diamonds in Fig. 6. This dataset has its largest connected component with $C_1 = 69.41\%$ and an average distance of $l_G = 7.35$. While the size of the largest component is twice that of `cnr-2000`, it has almost the same amount of vertices and half as many edges, with less dangling nodes (7.83% versus the 23.98% of `cnr-2000`). Adding to this, the average distance is much lower than that of `cnr-2000`). This explains the observation that `dblp-2010` achieved speedups higher than those of `cnr-2000` while at the same time being a less dense graph.

`eu-2005` speedup: speedups of around 3.00 were achieved (see the blue star and yellow cross markers in Fig. 7). These are parameter combinations which promote speed, only considering for the *hot vertex* set the vertices whose degree changed by at least 20%. The biggest component has size $C_1 = 87.26\%$ and the average distance is $l_G = 10.18$. This component size comprises almost the whole graph, with maximum (most influential vertex) $d_{out} = 6985$ and maximum (most influenced vertex) $d_{in} = 68922$. Compared to `cnr-2000`, this dataset has over two times as much vertices, yet the largest connected component jumps from 34.41% to 87.26%, meaning there is a much more dense and interconnected group of vertices in this graph. This explains the bigger speedups compared to the previous dataset.

`amazon-2008` speedup: in Fig. 8 we have speedups of around 3.00. Parameter combination ($r = 0.05, n = 2, \Delta = 1.00$) with the blue star plot had a stable speedup value close to this, only to be surpassed (starting from execution 30) by parameter combinations ($r = 0.20, n = 0, \Delta = 0.50$) and ($r = 0.20, n = 1, \Delta = 0.50$), represented by the yellow cross and the green diamond markers respectively. This dataset achieved the highest speedups out of all datasets. Its largest connected component has size $C_1 = 85.36\%$, similar to that of `eu-2005` (87.26%). The average distance $l_G = 12.06$ is close to that of `eu-2005` (10.18). The number of vertices $|V| = 735323$ is also close (`eu-2005` has 862664 vertices) but the number of edges $|E| = 5158388$ is less than a third of the other dataset (`eu-2005` has 19235140). The most influenced vertex in `amazon-2008` has $d_{in} = 1076$ and the most influential one has $d_{out} = 10$. Compared to `eu-2005`, this graph is less dense but at the same time the edges are much more spread out, leading to lower degrees. We attribute to this dynamic the similar speedups between `amazon-2008` and `eu-2005`.

Accuracy and scalability Figure 9 shows the observed accuracy (RBO) and size of summary graph (number of edges of the summary graph as a percentage of the number of edges of the complete graph) for datasets `eu-2005` and `amazon-2008`. Figures 10 (`eu-2005` and `amazon-2008`) and 11 (`hollywood-2011` and `eu-2015-host`) show the performance behaviour obtained with an increasing amount of parallelism. In those figures, *I/O time* corresponds to the time taken to ingest the updates from the stream; *Integration time* corresponds to the *I/O time* plus the time taken to apply updates to the graph through `Flink`'s internal dataflow-based execution; *Computation time* corresponds to the time taken to compute the algorithm.

`eu-2005`: We show the best three and worst three RBO parameter combinations in Fig. 9 (top-left). Parameters $r = 0.20, n = 0$ captured the highest RBO values. Investing purely on increasing n is not necessarily synonymous with achieving higher accuracy in a resource effective manner. The bottom-left of Fig. 9 shows that a value of $r = 0.05$ yielded a summary graph $\mathcal{G}$ with a number of edges around 75% of the original graph G (a higher n also contributes to increasing this value). The parameters with more conservative values (bigger n and lower r) led to the largest summary graphs. Figure 10 shows the results of isolating parameters $r = 0.05, n = 2, \Delta = 0.50$ (a balanced combination leaning towards better accuracy) to assess scalability with different worker counts. Speedups of around 10 were achieved with 16 workers (see top-left), with (mainly) computation, update integration and inherent I/O time benefiting from the increase in worker counts (bottom-left). These and further results illustrate that the VeilGraph implementation on top of `Flink` is scalable, since it does not introduce significant bottlenecks due to centralization of processing or coordination overhead. Furthermore, the larger the graph to be processed, the better speedups are expected as specific activity due to VeilGraph can be amortized over a larger-sized input.

`amazon-2008`: Fig. 9 (top-right) shows accuracy (RBO) maximized with parameters $r = 0.20, n = 0$ and different values of Δ, with a combination of $r = 0.05, n = 6$

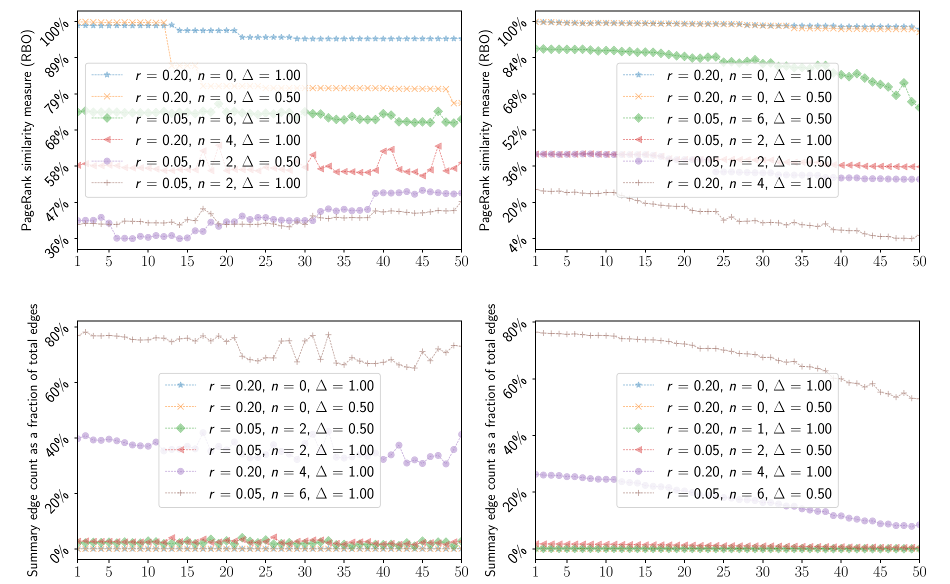


Fig. 9 Left-side `eu-2005`, right-side `amazon-2008`. First row shows the RBO values, second row shows the fraction of edges used by the summary graph

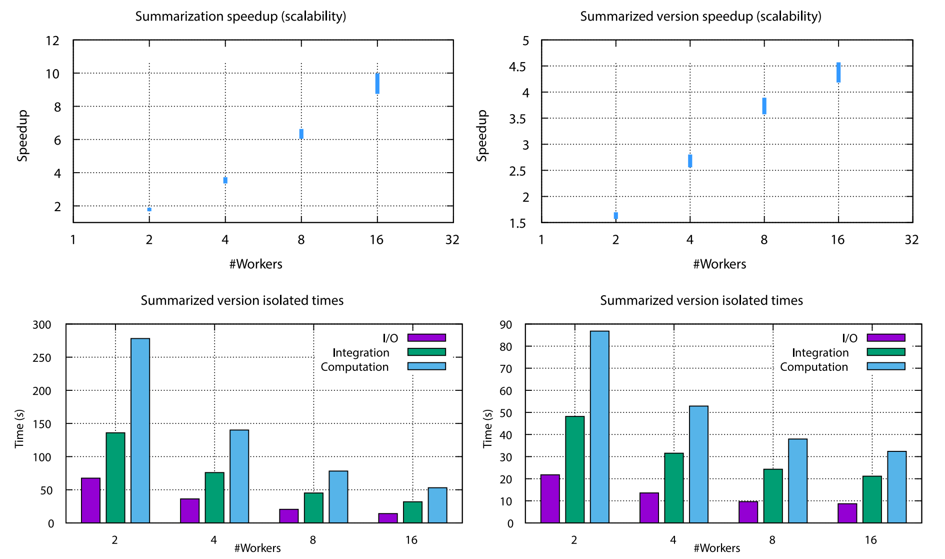


Fig. 10 Left-side `eu-2005`, right-side `amazon-2008`. First row shows the speedup for different numbers of workers, second row shows how time was distributed between computation and integration of the updates

achieving slightly lower accuracy with a more accentuated error accumulation. We attribute the observed behaviour of lower RBO with a much higher n to the impact of the edge removal on the topology of the `amazon-2008` dataset. The number of summary graph edges as a fraction of the complete graph's edges is in line with previous results: greater values of n led to a larger summary graph—see Fig. 9 (bottom-right). Still, there was a pattern with $n = 6$ where there was a tendency for RBO to decrease

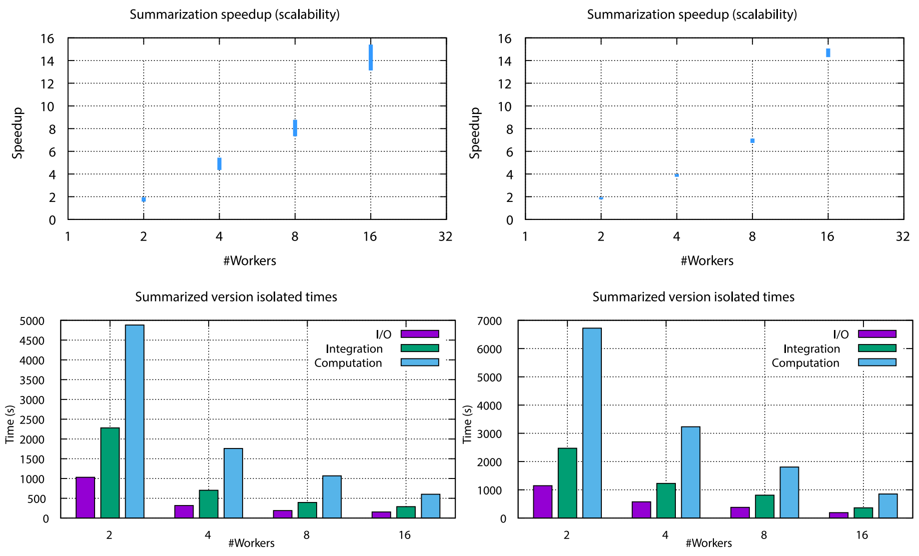


Fig. 11 Left-side `hollywood-2011`, right-side `eu-2015-host`. First row shows the speedup for different numbers of workers, second row shows how time was distributed between computation and integration of the updates

(green diamond marker) coupled with a tendency for the summary graph edge fraction to decrease (brown + marker) too.

Fig. 10 (top-right) shows that the scalability was lower than that of the `eu-2005` dataset as the number of workers is increased. This difference can be explained by the fact that `amazon-2008` has almost one fourth of the edges present in `eu-2005` (the former has more edges to process and hence to benefit from our model). The way time is distributed between I/O, integration and computation (bottom-right) is similar to `eu-2005`.

`hollywood-2011`: This social network has about 45x more edges than `amazon-2008` (around 229 M edges). We evaluated this larger dataset to focus on the scalability of our model. Figure 11 shows (top-left) that the speedup of the computational time was close to linear. Like before, the obtained computational speedups do not include I/O and integration time as we are focused on highlighting that the scalability of the model itself—the larger the graph, the greater the benefit that our summary graph model will have from more workers.

`eu-2015-host`: This web graph has around 386 M edges and the obtained speedups over the computation are show in Fig. 11 (top-right). Similarly to `hollywood-2011`, this dataset achieved high speedups, reinforcing the link between the model's increased efficiency and the greater-sized graphs.

Analysis and discussion The speedups are indicative that as we test with larger datasets, executing a graph algorithm over just VeilGraph's K set instead of the complete graph is beneficial to performance—as previously mentioned, the larger the graph, the greater the benefits of our model.

VeilGraph is able to achieve faster execution while maintaining very competitive levels of accuracy, such as the case of parameters ($r = 0.20, n = 0, \Delta = 0.50$), achieving over 50% faster computational time with an RBO above 90%. Such reduction in latency, without significant loss in accuracy for application purposes, can be very relevant in cloud scenarios where reduced latency (and the enabled reductions to the bounds of SLA latency percentiles) is often more relevant than the raw throughput of the system per se (e.g., [37, 38]). If necessary, VeilGraph allows further reductions in latency (up to 100x faster than the baseline full graph processing *strawman* competing alternative), but naturally at the cost of accuracy, e.g., by means of enforcing only lower levels of RBO.

We note that it would be interesting to further evaluate `hollywood-2011` and `eu-2015-host` with even bigger cluster sizes to analyse potentially-diminishing returns in light of theoretical models for assessing scalability [39]. Despite this, our method has achieved a very good trade-off between result accuracy and reduction in total computation (be it in number of processed graph elements or direct time comparison), bearing in mind again the deliberate lower-bound performance context we are employing in our assessment (recall: we are executing and comparing with the full processing of the graph just on 50 instants, against doing it on every vertex/edge update).

Randomized edge removals were used to assess the robustness of the model. Removals have an impact on graph topology which does not necessarily manifest as one would expect. Extending the computational scope by using conservative parameters in VeilGraph overall promoted a greater accuracy in our experiments, but special consideration must be given to the cases where removals may lead to a cascading effect, where extending the model to bound error propagation is an interesting challenge.

The visualization of separate computational times as shown in Fig. 11 is revealing of a bi-dimensional cost that is often overlooked in these distributed computational platforms. This cost manifests, as one would expect, in the overheads of communication between cluster elements and the writing/reading of data from distributed storage. Its second dimension (also a trait in `Apache Spark`) is located in the runtime of `Flink` itself. The user-logic (in our case written in `Java`) for the most part is able to abstract away that the underlying execution will take place in a distributed system. However, unoptimized, this incurs relevant communication costs which are not always obvious and whose measurement requires breaking down this abstraction. This may be achieved by directly measuring the metrics of operators executed as part of the optimized execution plan produced by the `Flink` compiler and is something we plan to explore in future work. Another interesting observation is that with more than 8 workers, for the smaller `eu-2005, amazon-2008`, the speedups obtained by horizontally-scaling the cluster diminished faster in the summary model than with using the complete computations. This is due to the summary version processing much fewer elements and thus not leveraging extra workers so much. As we tested with the bigger datasets `hollywood-2011` and `eu-2015-host`, the benefits of horizontal scaling on our model increased dramatically.

Related work

Our multidisciplinary work encompasses paradigms to express graph computations, stream processing and approximation techniques. We present relevant state-of-the-art contributions that address these dimensions and comment on how VeilGraph also addresses them with respect to each model:

- `Kineograph` is a distributed system designed to capture the relations in incoming data feeds [40], built to maintain timely updates against a continuous flux of data. The architecture of `Kineograph` considers two types of working distributed system nodes: *ingest* nodes which are responsible for registering graph update operations as identifiable transactions, to be distributed to *graph* nodes. This latter type of node forms a key/value store that is distributed in memory. `Kineograph` performs computation on static snapshots, simplifying the design of algorithms. The design of VeilGraph goes beyond this design by extending its concept to give users the flexibility to design algorithms for either the complete graph or summarized versions, with little difference. Users can incorporate the awareness that the graph has changed, or opt to design an algorithm that considers the current graph as a static version, like `Kineograph`. VeilGraph is implemented over `Flink`, a generic dataflow programming framework with a graph processing library, `Gelly`.

  `Kineograph` has an architecture which attributes importance to the task of registering new graph information and the task of storing it. Our architecture's design provides extra flexibility to programmers as it is implemented over a generic programming model compared to `Kineograph`. However, as VeilGraph uses `Flink` which does not consider the explicit storage of the graph and its ingestion in design of the workers in a cluster of machines, it could certainly benefit from implementing these concerns directly as a `Flink` module.
- `KickStarter` showcased a technique for trimming approximate values of vertex subsets which were impacted by edge deletion [41].
  `KickStarter` deals with edge deletions by identifying values impacted by the deletions and adapting the network impacts before the following computation, achieving good results on real-world use-cases. By focusing on monotonic graph algorithms, its scope is specific only to selection-based algorithms. We decouple in VeilGraph the approximation technique, the summarization model and the algorithm type. Thus, we are able to offer the big vertex model and provide a structured sequence of steps to integrate another model or approximation technique (e.g. `KickStarter`'s own technique could be a candidate). Rather than the hard-coded logic for bounding the approximation values specifically to monotonic algorithms of `KickStarter`, we offer in VeilGraph a middleware layer which offers additional customization capabilities to programmers, for example, to implement a different logic to bound approximations or to even implement automated strategies based on statistical analyses and machine learning.[8]
- `Tornado` is a system with an asynchronous bounded iteration model, offering fine-grained updates while ensuring correctness [42]. It is based on the observations that: *1)* loops starting from *good enough* guesses usually converge quickly; *2)* for many iterative methods, the running time is closely relative to the approximation error. Whenever a result request is received, the model creates a branch loop from the main loop. This branch loop uses the most recent approximations as a guess for the algorithm. It is a technique that could benefit from applying VeilGraph 's summarization model,

[8] E.g., as done in previous work targeting workflows of Map-Reduce jobs [61].

as Tornado's main loop could produce approximations faster, making the computation result guesses readily available as queries arrive.

- Naiad is a dataflow processing system [43] offering different levels of complexity and abstractions to programmers. It allows programmers to implement graph algorithms such as weakly connected components, approximate shortest paths and others while achieving better performance than other systems. Naiad allows programmers to use common high-level APIs to express algorithm logic and also offers a low-level API for performance. Its concepts are important and other systems could benefit from offering tiered programming abstraction levels as in Naiad. Its low-level primitives allow for the combination of dataflow primitives (similar to those VeilGraph uses from Flink) with finer-grained control on iterative computations. An extension to Flink's architecture to offer this detailed control would enrich the abilities that our framework is able to offer to users.
- GraphBolt [44] is a recent work building on KickStarter, offering a generalized incremental programming model enabling the development of incremental versions of complex aggregations. They evaluate different algorithms while defining different aggregation functions for each in order to support the computation approximations. This system is very relevant as their work focuses on crafting functions for specific use-cases, though with a reduction in control for the programmer, as there is no way to provide custom-defined logic for how dependency tracking and error tolerance is defined.
- FlowGraph [45] is a system that proposes a syntax for a language to detect temporal patterns in large-scale graphs and introduces a novel data structure to efficiently store results of graph computations. This system is a unification of graph data with stream processing considering the changes of the graph as a stream to be processed and offering an API to satisfy temporal patterns. FlowGraph's model for addressing the temporal evolution of the graph and subsequent execution is innovative by formalizing the pattern detection with its own language approach. Integrating this approach in the post-update pre-execution stage of VeilGraph could provide a richer model for the programmer.

We also go over other systems and architectures for graph processing. Although they do not address these dimensions directly, they were candidate open-source platforms to implement the VeilGraph model.

- Apache Giraph [46]. An open-source implementation of Google Pregel [32], tailor-made for graph algorithms. It was created as an efficient and scalable fault-tolerant implementation on clusters with thousands of commodity hardware, hiding implementation details underneath abstractions. Work has been done to extend Giraph from the *think-like-a-vertex* model to *think-like-a-graph* [47]. It uses Apache Hadoop's MapReduce implementation to process graphs. It was inspired by the *Bulk-Synchronous-Parallel (BSP)* model. Apache Giraph allows for master computation, sharded aggregators, has edge-oriented input, and also uses out-of-core computation – limited partitions in memory. Partitions are stored in local disks, and for cluster computing settings, the out-of-core partitions are spread out across all disks. Giraph's authors use the concept of *superstep*, which are sequences of iterations for graph processing. In a superstep S, a user-supplied function is executed for each vertex v (this can be done in parallel) that has a status of active. When S terminates, all vertices may send messages which can be processed by user-defined functions at step $S+1$. Giraph attempts to keep vertices and edges in memory and uses only the network for the transfer of messages. As far as we know, it is not being currently actively developed.

- Apache Spark and its GraphX [48] graph processing library. It is a graph processing framework built on top of Apache Spark, enabling low-cost fault-tolerance. The authors target graph processing by expressing graph-specific optimizations as distributed join optimizations and graph views' maintenance. In GraphX, the property graph is reduced to a pair of collections. This way, the authors are able to compose graphs with other collections in a distributed dataflow framework. Operations such as adding additional vertex properties are then naturally expressed as joins against the collection of vertex properties. Graph computations and comparisons are thus an exercise in analysing and joining collections. GraphX is to Spark what Gelly is to Flink. They are both graph processing libraries implemented over distributed processing frameworks. Flink's library offers many more graph algorithms compared to GraphX. Considering this and the fact that Flink has seen growing adoption in the industry recently, we considered Flink to be a stronger candidate compared to Spark.
- Chaos [49] is an evolution of X-Stream [50]. On top of the secondary storage studies performed in the past, graph processing in Chaos achieves scalability with multiple machines in a cluster computing system. It is based on different functionalities: load balancing, randomized work stealing, favouring sequential access to storage and an adaptation of X-Stream's streaming partitions to enable parallel execution. Chaos uses an edge-centric *Gather-Apply-Scatter (GAS)* model of computing. It is composed of a storage sub-system and a computation sub-system. The former exists concretely as a storage engine in each machine. Its concern is that of providing edges, vertices and updates to the computation sub-system. Previous work on X-Stream highlighted that the primary resource bottleneck is the storage device bandwidth. In Chaos, the storage and computation engines' communication is designed in a way that storage devices are busy all the time – thus optimizing for the bandwidth bottleneck. Chaos, though it certainly has merits, it is a platform not as widely adopted in academic and industry as Flink and Spark. For that reason, though its idea is useful, we did not choose it to implement and evaluate our model.

Conclusion

Graph processing is a cornerstone of modern data processing, presenting different dimensions which grow in complexity together with their relevance. The relationships between these areas of graph processing are crucial to design better techniques and tools to process big graphs in the future [51]. Focusing on approximate graph processing as a technique to speed up stream-based graph change integration and graph algorithm result updates, we designed VeilGraph. It provides a model and architecture to incorporate custom approximate processing strategies and to enable choosing between built-in behaviours.

Effectively, as graphs are growing in size, the need to process them more efficiently also becomes crucial. Working with dynamic graphs, there are many use-cases where frequent updates do not change the graph entirely. In this context, many elements of the graph may remain unchanged, creating the possibility to reduce computation by delaying or prioritizing computation over specific parts of the graph, enabling more efficient processing.

Among these use-cases, we can consider for example the analysis of social networks. Graphs such as those of Facebook, Twitter and WhatsApp benefit from ranking users. Considering their magnitude, and that overall these graphs evolve frequently (with creation of new vertices and edges more frequent than deletion), updating a view on the relative importance of users (vertices) can directly harness the performance optimization of VeilGraph.

This work touches upon concepts in the literature, of which the systems mentioned in "Related work" section are but a (very important) component. Approximate processing may manifest in several forms, and it shares a close relationship with stream processing. There exist works in the literature which focus on maintaining approximations over streams, accounting for different aspects. In the initial approaches to stream processing, one may find algorithms for updating statistics over streams using sliding windows [52] (there exist different stream window models) using for example randomization algorithms [53] and sampling approaches [23, 54]. Moving to graph-based data problems, stream processing gains specific challenges which have been analysed [55], with contributions to the state-of-the-art which explored techniques to create graph representations which allow the update of graph metrics, using structures such as sparsifiers, spanners and sub-graphs [26]. Stream processing applied to graphs has many specificities such as the nature of dynamism of the graph [56], different graph algorithm results to maintain [57] and also becomes more complex as the size of data increases [25].

Approximate processing also bridges into important avenues of computation optimization. There are motivations to explore approximation of results. It is worthy to explore the relationship between result accuracy and computation performance as not all scenarios require absolute accuracy. This relationship may manifest for example in distributed processing contexts, where as an example not all parallel tasks may finish in the same time. Techniques exist to mitigate this, such as dropping straggler tasks [58] (e.g., when studying the performance of these techniques in `MapReduce` [22]).

This work focused on the overlap of both of these concerns—a stream processing context over a distributed computation platform (such as `Apache Flink`). Our model was evaluated in the context of random walk problems, for which the observed results lead us to conclude that the VeilGraph model, even when tested in a deliberately challenging context for comparability, is a viable basis to enable faster, more efficient and configurable graph processing on this type of problem in many real-world scenarios. The results we obtain were produced under a sensible and realistic streaming scenario where graph updates are large enough so that the changes to the graph would not explicitly benefit (and almost in an unbounded way) the summarization model we proposed and evaluated. The design herein presented focused on locality of graph update impacts to heuristically process only the graph elements which are more directly impacted and with greater magnitude.

VeilGraph 's model has relevance for practitioners and business tasks which depend on constantly obtaining updated information from graph-based data. Vertex centrality, the type of problem used to evaluate our idea, is important as an enabler of ranking techniques which are essential to assess the more important vertices in networks. Domains such as social network analysis [59], recommendation systems and even leader election for service placement [20] stand to benefit from the trade-offs between accuracy and performance which are enabled through VeilGraph.

Future work

While the programming and computation model presented in VeilGraph has an algorithm-agnostic structure, for the purpose of evaluation it was evaluated with vertex centrality, which although relevant for many purposes such as processing big graph data [60], does not represent all cases. Considering this, we aim to add new techniques and models for problems like online community finding and to perform further evaluations on larger datasets. Extending the big vertex summary use-case to explore its application in representing communities (potentially with multiple instances of big vertices) is a scenario to explore.

Another direction of future work consists in enriching the model with error management findings from other contributions (e.g., `GraphBolt` [44]) and also implementing the VeilGraph model on other platforms (or updated versions of the already considered ones, in the future) could pave the way for new validations and extensions of the model. We plan to further research the challenge of the disruptive aspects of edge deletions over graph topology and how that may speed up or slow down the propagation of approximation errors. We are researching different approximation strategies based on the statistical records, from a set of manually implemented policies to automation based on statistics.

Acknowledgements
This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects UIDB/50021/2020, PTDC/EEI-COM/30644/2017, and PTDC/CPO-CPO/28495/2017.

Authors' contributions
MC contributed with the implementation of VeilGraph in `Flink` and the evaluation of of the proposed middleware. MC, SE and LV participated in iterations of the design of the VeilGraph model and architecture. AF and LV participated in the validation of the model. MC, AF, LV participated in the analysis of the related work. All authors read and approved the final manuscript.

Funding
Fundação para a Ciência e a Tecnologia, under projects UIDB/50021/2020, PTDC/EEI-COM/30644/2017, and PTDC/CPO-CPO/28495/2017.

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 7 September 2021 Accepted: 29 December 2021
Published online: 23 February 2022

References

1. Coimbra ME, Francisco AP, Veiga L. An analysis of the graph processing landscape. J Big Data. 2021;8(1):55. https://doi.org/10.1186/s40537-021-00443-9.
2. Chung F. Graph theory in the information age. Notice AMS. 2010;57(6):726–32.
3. Meusel R, Vigna S, Lehmberg O, Bizer C. The graph structure in the web-analyzed on different aggregation levels. J Web Sci. 2015;1:89.
4. Lim J, Ryu S, Park K, Choe YJ, Ham J, Kim WY. Predicting drug-target interaction using a novel graph neural network with 3D structure-embedded graph representation. J Chem Inf Model. 2019;59(9):3981–8.
5. Liang S, Stockinger K, de Farias TM, Anisimova M, Gil M. Querying knowledge graphs in natural language. J Big Data. 2021;8(1):1–23.
6. Donnelly Gordon. 75 Super-Useful Facebook Statistics for 2018. Accessed 5 May 2020. 2020.
7. Pho P, Mantzaris AV. Regularized simple graph convolution (sgc) for improved interpretability of large datasets. J Big Data. 2020;7(1):1–17.
8. Sassi I, Anter S, Bekkhoucha A. A graph-based big data optimization approach using hidden markov model and constraint satisfaction problem. J Big Data. 2021;8(1):1–29.
9. Chinazzi M, Davis JT, Ajelli M, Gioannini C, Litvinova M, Merler S, Piontti AP, Mu K, Rossi L, Sun K, et al. The effect of travel restrictions on the spread of the 2019 novel coronavirus (covid-19) outbreak. Science. 2020;9:78.
10. Maduako I, Wachowicz M, Hanson T. Stvg: an evolutionary graph framework for analyzing fast-evolving networks. J Big Data. 2019;6(1):1–24.
11. Chowdhury S, Khanzadeh M, Akula R, Zhang F, Zhang S, Medal H, Marufuzzaman M, Bian L. Botnet detection using graph-based feature clustering. J Big Data. 2017;4(1):1–23.
12. Junghanns M, Petermann A, Teichmann N, Gómez K, Rahm E. Analyzing extended property graphs with apache flink. In: Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics. NDA '16. 2016. New York: ACM. p. 3–138. https://doi.org/10.1145/2980523.2980527.
13. Langville AN, Meyer CD. Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton: Princeton University Press; 2011.
14. Freeman LC. A set of measures of centrality based on betweenness. Sociometry. 1977;40(1):35–41.
15. Katz L. A new status index derived from sociometric analysis. Psychometrika. 1953;18(1):39–43. https://doi.org/10.1007/BF02289026.
16. Newman M. Networks: An Introduction. New York: Oxford University Press Inc; 2010.
17. Vassilevich DV. Heat kernel expansion: user's manual. Phys Rep. 2003;388(5–6):279–360.
18. Boldi P, Rosa M, Santini M, Vigna S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Srinivasan S, Ramamritham K, Kumar A, Ravindra MP, Bertino E, Kumar R, (eds.) Proceedings of the 20th International Conference on World Wide Web. ACM: New York. 2011. p. 587–596
19. Chung F, Simpson O. Distributed algorithms for finding local clusters using heat kernel pagerank. In: International Workshop on Algorithms and Models for the Web-Graph, pp. 177–189. Springer: Cham. 2015.
20. Coimbra ME, Selimi M, Francisco AP, Freitag F, Veiga L. Gelly-scheduling: distributed graph processing for service placement in community networks. In: Haddad HM, Wainwright RL, Chbeir R, eds. Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018. ACM: New York; 2018. p. 151–160. https://doi.org/10.1145/3167132.3167147.
21. Agarwal S, Milner H, Kleiner A, Talwalkar A, Jordan M, Madden S, Mozafari B, Stoica I. Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. SIGMOD '14. New York: ACM. p. 481–492. 2014. https://doi.org/10.1145/2588555.2593667.
22. Goiri I, Bianchini R, Nagarakatte S, Nguyen TD. Approxhadoop: Bringing approximations to mapreduce frameworks. SIGARCH Comput Archit News. 2015;43(1):383–97. https://doi.org/10.1145/2786763.2694351.
23. Babcock B, Datar M, Motwani R. Sampling from a moving window over streaming data. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '02, pp. 633–634. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 2002. http://dl.acm.org/citation.cfm?id=545381.545465.
24. Hu P, Lau WC. A Survey and Taxonomy of Graph Sampling. 2013. arXiv:1308.5865.
25. Ahmed NK, Duffield N, Willke TL, Rossi RA. On sampling from massive graph streams. Proc VLDB Endow. 2017;10(11):1430–41. https://doi.org/10.14778/3137628.3137651.
26. Ahn KJ, Guha S, McGregor A. Graph sketches: Sparsification, spanners, and subgraphs. In: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. PODS '12. ACM: New York. 2012. p. 5–14. https://doi.org/10.1145/2213556.2213560.
27. Chien S, Dwork C, Kumar R, Simon DR, Sivakumar D. Link Evolutions: Analysis and Algorithms. Internet Math. 2003;1(3):277–304.
28. Babcock BB, Datar M, Motwani R, Mayur BB, Babcock BB, Datar M, Motwani R. Load Shedding Techniques for Data Stream Systems. In: In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS, 2003;pp. 1–3. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.1941.
29. Kalavri V, Simas T, Logothetis D. The shortest path is not always a straight line: leveraging semi-metricity in graph analysis. Proc VLDB Endowment. 2016;9(9):672–83.
30. Langville AN, Meyer CD. Updating pagerank with iterative aggregation. In: Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers&Amp; Posters. WWW Alt. '04, pp. 392–393. ACM, New York, NY, USA. 2004. https://doi.org/10.1145/1013367.1013491. http://doi.acm.org/10.1145/1013367.1013491.

31. Kalavri V, Ewen S, Tzoumas K, Vlassov V, Markl V, Haridi S. Asymmetry in large-scale graph analysis, explained. In: Proceedings of Workshop on GRAph Data Management Experiences and Systems. GRADES'14. New York: ACM. p. 4–147. 2014. https://doi.org/10.1145/2621934.2621940.
32. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: A System for Large-scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD '10, pp. 135–146. New York: ACM. 2010. https://doi.org/10.1145/1807167.1807184.
33. Page L, Brin S, Motwani R, Winograd T. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab. 1999. http://ilpubs.stanford.edu:8090/422/.
34. Boldi P, Vigna S. The WebGraph framework I: Compression techniques. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) Proceedings of the 13th International Conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004. New York: ACM. p. 595–602; 2004. https://doi.org/10.1145/988672.988752.
35. Webber W, Moffat A, Zobel J. A similarity measure for indefinite rankings. ACM Trans Inf Syst. 2010;28(4):20–12038. https://doi.org/10.1145/1852102.1852106.
36. Moffat A. Computing maximized effectiveness distance for recall-based metrics. IEEE Transa Knowl Data Eng. 2018;30(1):198–203.
37. Reda W, Canini M, Suresh L, Kostić D, Braithwaite S. Rein: Taming tail latency in key-value stores via multiget scheduling. In: Proceedings of the Twelfth European Conference on Computer Systems. EuroSys '17, pp. 95–110. Association for Computing Machinery, New York, NY, USA. 2017. https://doi.org/10.1145/3064176.3064209.
38. Misra PA, Borge MF, Goiri In, Lebeck AR, Zwaenepoel W, Bianchini R. Managing tail latency in datacenter-scale file systems under production constraints. In: Proceedings of the Fourteenth EuroSys Conference 2019. EuroSys '19. Association for Computing Machinery, New York, NY, USA. 2019. https://doi.org/10.1145/3302424.3303973.
39. Gustafson JL. Gustafson's Law, pp. 819–825. Springer, Boston. 2011. https://doi.org/10.1007/978-0-387-09766-4_78.
40. Cheng R, Hong J, Kyrola A, Miao Y, Weng X, Wu M, Yang F, Zhou L, Zhao F, Chen E. Kineograph: Taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems. EuroSys '12, pp. 85–98. ACM, New York, NY, USA. 2012. https://doi.org/10.1145/2168836.2168846.
41. Vora K, Gupta R, Xu G. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '17, pp. 237–251. ACM, New York, NY, USA. 2017. https://doi.org/10.1145/3037697.3037748.
42. Shi X, Cui B, Shao Y, Tong Y. Tornado: A system for real-time iterative analysis over evolving data. In: Proceedings of the 2016 International Conference on Management of Data. SIGMOD '16, pp. 417–430. ACM, New York, NY, USA. 2016. https://doi.org/10.1145/2882903.2882950.
43. Murray DG, McSherry F, Isaacs R, Isard M, Barham P, Abadi M. Naiad: A timely dataflow system. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13, pp. 439–455. ACM, New York, NY, USA. 2013. https://doi.org/10.1145/2517349.2522738.
44. Mariappan M, Vora K. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In: Proceedings of the Fourteenth EuroSys Conference 2019. EuroSys '19, pp. 25–12516. ACM, New York. 2019. https://doi.org/10.1145/3302424.3303974.
45. Chaudhry HN. Flowgraph: Distributed temporal pattern detection over dynamically evolving graphs. In: Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, 2019; p. 272–275.
46. Ching A. Scaling apache giraph to a trillion edges. Facebook Engineering Blog; 2013. p. 25.
47. Tian Y, Balmin A, Corsten SA, Tatikonda S, McPherson J. From "think like a vertex" to "think like a graph". Proc VLDB Endow. 2013;7(3):193–204. https://doi.org/10.14778/2732232.2732238.
48. Xin RS, Gonzalez JE, Franklin MJ, Stoica I. Graphx: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems. GRADES '13, pp. 2–126. ACM, New York, NY, USA. 2013. https://doi.org/10.1145/2484425.2484427.
49. Roy A, Bindschaedler L, Malicevic J, Zwaenepoel W. Chaos: Scale-out graph processing from secondary storage. In: Proceedings of the 25th Symposium on Operating Systems Principles. SOSP '15, pp. 410–424. ACM, New York, NY, USA. 2015. https://doi.org/10.1145/2815400.2815408. http://doi.acm.org/10.1145/2815400.2815408.
50. Roy A, Mihailovic I, Zwaenepoel W. X-stream: Edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13, pp. 472–488. ACM, New York, NY, USA. 2013. https://doi.org/10.1145/2517349.2522740. http://doi.acm.org/10.1145/2517349.2522740.
51. Sakr S, Bonifati A, Voigt H, Iosup A, Ammar K, Angles R, Aref W, Arenas M, Besta M, Boncz PA, Daudjee K, Valle ED, Dumbrava S, Hartig O, Haslhofer B, Hegeman T, Hidders J, Hose K, Iamnitchi A, Kalavri V, Kapp H, Martens W, Özsu MT, Peukert E, Plantikow S, Ragab M, Ripeanu MR, Salihoglu S, Schulz C, Selmer P, Sequeda JF, Shinavier J, Szárnyas G, Tommasini R, Tumeo A, Uta A, Varbanescu AL, Wu H-Y, Yakovets N, Yan D, Yoneki E. The future is big graphs: A community view on graph processing systems. Commun ACM. 2021;64(9):62–71. https://doi.org/10.1145/3434642.
52. Datar M, Gionis A, Indyk P, Motwani R. Maintaining stream statistics over sliding windows: (extended abstract). In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '02, pp. 635–644. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 2002. http://dl.acm.org/citation.cfm?id=545381.545466.
53. Arasu A, Manku GS. Approximate counts and quantiles over sliding windows. In: Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2004. Association for Computing Machinery, Inc., ??? 2004. https://www.microsoft.com/en-us/research/publication/approximate-counts-and-quantiles-over-sliding-windows/.
54. Vitter JS. Random sampling with a reservoir. ACM Trans Math Softw. 1985;11(1):37–57. https://doi.org/10.1145/3147.3165.
55. Feigenbaum J, Kannan S, McGregor A, Suri S, Zhang J. On graph problems in a semi-streaming model. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings. Lecture Notes in Computer Science, vol. 3142, pp. 531–543. Springer, ??? 2004. https://doi.org/10.1007/978-3-540-27836-8_46.

56. Besta M, Fischer M, Kalavri V, Kapralov M, Hoefler T. Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. arXiv preprint arXiv:1912.12740 2019.
57. Kalavri V, Carbone P, Bali D, Abbas Z. Gelly Streaming. [Online, GitHub; accessed 24-April-2020] 2019. https://github.com/vasia/gelly-streaming.
58. Ananthanarayanan G, Hung MC-C, Ren X, Stoica I, Wierman A, Yu M. Grass: Trimming stragglers in approximation analytics. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pp. 289–302. USENIX Association, Seattle, WA. 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/ananthanarayanan.
59. Al-Molhem NR, Rahal Y, Dakkak M. Social network analysis in telecom data. Journal of Big Data. 2019;6(1):99.
60. Zhang H, Raitoharju J, Kiranyaz S, Gabbouj M. Limited random walk algorithm for big graph data clustering. Journal of Big Data. 2016;3(1):1–22.
61. Esteves S, Galhardas H, Veiga L. Adaptive execution of continuous and data-intensive workflows with machine learning. In: Ferreira, P., Shrira, L. (eds.) Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10–14, 2018, pp. 239–252. New York: ACM; 2018. https://doi.org/10.1145/3274808.3274827.

Publisher's Note