Zooming Out on an Evolving Graph

Amir Aghasadeghi
New York University
amirpouya@nyu.edu

Vera Z. Moffitt
Drexel University
zaychik@drexel.edu

Sebastian Schelter
New York University
ss12727@nyu.edu

Julia Stoyanovich[*]
New York University
stoyanovich@nyu.edu

ABSTRACT

An evolving graph maintains the history of changes of graph topology and attribute values over time. Such a graph has a specific temporal and structural resolution. It is often useful to modify this resolution during analysis, for example, to consider communities rather than individual nodes, or to quantify changes at the level of days rather than hours.

We propose *attribute-based zoom* and *temporal window-based zoom* — two operators that support exploratory analysis of an evolving graph at different levels of resolution. We develop several alternative physical representations of an *evolving property graph* — a temporal generalization of a property graph — and detail how to implement the proposed zoom operators using dataflow operations. These different physical representations allow us to explore the trade-offs in temporal and structural locality with respect to the performance of the zoom operators. We implement the operators in Apache Spark, evaluate them on real evolving graph datasets, and demonstrate scalability to billion-edge graphs.

1 INTRODUCTION

Many social structures and systems can be represented as networks or graphs. The phenomena that are represented by these graphs can change over time, and therefore, many interesting questions about these graphs are related to their evolution rather than to their static state. Researchers study graph evolution rate and mechanisms [1, 9], the impact of specific events on further evolution [8, 39] and spatio-temporal patterns [27, 28], with most progress taking place in the last decade [24, 35, 37, 38, 40].

Our focus in this paper is on a temporal generalization of a property graph, called TGraph, which we recently introduced [37]. Figure 1 shows an example — an interaction network in which nodes represent people, and, for the students among them, include information about a school at which they are enrolled, while edges represent co-authorship. As in conventional property graphs [3], nodes and edges of a TGraph are associated with a set of key-value pairs that represent an assignment of values to attributes. In addition, TGraph associates a time interval (representing a set of discrete consecutive time points) with each state of a node or edge. For example, a `person` node *Ann* exists, and is enrolled at MIT, during the interval $T = [1, 7)$.

TGraph maintains the history of changes of graph topology and attribute values over time. It has a specific *temporal* and *structural resolution*, which users often want to modify for exploratory analysis, for example, to look at communities rather than individual nodes, or to quantify changes at the level of days rather than hours. In this paper we focus on two operators, aZoom^T and wZoom^T, that allow us to change the structural and temporal resolution of a TGraph, respectively. These operators are part of a compositional evolving graph algebra called TGA, which we presented in [37], that operates under *point semantics* [5].[1] A consequence of these semantics is that the TGraph must remain *temporally coalesced* — vertices and edges in the output of an operator must be associated with time periods of maximal length during which no change occurred.

Attribute-based zoom (aZoom^T). We may be interested in analyzing evolving graphs at different levels of *structural resolution*, to study properties and behavior of individual nodes, of communities, and of the graph as a whole. An operation that achieves this, known as *node creation*, is present in several conventional (non-temporal) graph query languages [14, 21, 32, 42]. Our focus is on a temporal generalization of this operation for graphs, called *temporal attribute-based zoom*, or aZoom^T for short.

Consider TGraph $\mathcal{G}_1$ in Figure 1, where school names are represented as values of the school property of person nodes. aZoom^T computes the TGraph in Figure 2, where schools become nodes (actors) rather than values.

aZoom^T is evaluated over a TGraph under *point semantics* and, specifically, under the principle of snapshot reducibility [5]: we evaluate the non-temporal variant of the operator over each state of the graph (also known as a "snapshot"), and then apply temporal coalescing [4] to represent each vertex or edge in the result with a single fact, corresponding to the longest interval during which no change occurred. aZoom^T is described in Section 2.2.

Temporal window-based zoom (wZoom^T). This operator changes the *temporal resolution* of a TGraph. This operation is important because it may not be known *a priori*, at the time when graph evolution is being recorded, at what time scale interesting trends can be observed. For example, changes in node centrality in a social network may be observable on the scale of weeks but not months. Understanding at what temporal resolution to consider network evolution is an integral part of exploratory analysis. Let us return to our running example in Figure 1, and assume that time points represent months of 2019. We may zoom out on $\mathcal{G}_1$ temporally, to 3-month windows, retaining nodes and edges in the result for a particular time window that were present in the input during all time points of the window. The result is presented in Figure 3, and described in more detail in Section 2.3.

Next, we explore different physical representations to answer the following questions: (i) How should we represent a TGraph to compute the result of aZoom^T and wZoom^T efficiently? Should we use a snapshot-based representation, storing graph evolution as a sequence of conventional graphs, that is easy to parallelize but lacks compactness, or should we leverage a more compact representation, as suggested by Figure 1? (ii) What representation should we use to efficiently execute *a sequence* of these operators? We address these questions, making the following contributions:

- We propose different physical representations of a TGraph and detail how to define aZoom^T and wZoom^T using dataflow operations for these representations (Section 3).

[*]This work was supported in part by NSF Grant No. 1916505.

[1]The focus of [37] is on defining the TGraph model and algebra, while this paper focuses on system and implementation aspects.

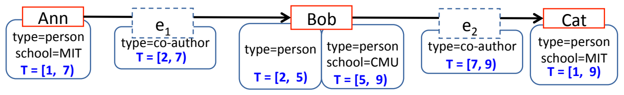


Figure 1: Evolving property graph (TGraph) $\mathcal{G}_1$.

- We describe how to efficiently implement aZoom^T and wZoom^T in Apache Spark (Section 4).
- We conduct an extensive experimental evaluation of aZoom^T and wZoom^T and demonstrate scalability to billion-edge graphs. We find that a physical representation that balances temporal and structural locality outperforms other representations in most cases (Section 5).

2 TGRAPH MODEL AND ZOOM OPERATORS

We provide the background on the evolving property graph model called TGraph, and define the operators aZoom^T and wZoom^T that take a valid TGraph as input, and output a TGraph.

2.1 Evolving property graphs

In [37] we proposed a logical model of an evolving graph called TGraph that represents a single graph (such as the Web, or a large collaboration network), and models the evolution of its topology, and vertex and edge properties. A TGraph is a directed multigraph: its nodes and edges have identity, and multiple edges may connect a given pair of nodes. Each entity (node and edge) has a required type label, and is associated with a (possibly empty) set of key-value pairs that represent its properties, each in the form of a property label (key) and a corresponding value. The set of properties for an entity is not fixed: it can be different among entities of the same type, and for the same entity over time.

We now recall the definition of TGraph from [37], simplifying it slightly. This definition extends the static property graph definition of Angles et al. [3] by associating periods of validity with graph nodes, edges, and their properties. Time is drawn from a linearly ordered discrete domain Ω^T.

Definition 2.1. A TGraph $\mathcal{G} = (V, E, L, \rho, \xi^T, \lambda^T)$ is a six-tuple:

- V is a finite set of *nodes* (or *vertices*), E is a finite set of *edges*, $V \cap E = \emptyset$, and L is a finite set of property labels;
- $\rho : E \longrightarrow (V \times V)$ is a total function that maps an edge to its source and destination nodes;
- $\xi^T : (V \cup E) \times \Omega^T \longrightarrow B$ is a total function that maps a node or an edge and time point to a Boolean, indicating existence of the node or edge at that time point; and
- $\lambda^T : (V \cup E) \times L \times \Omega^T \longrightarrow val$ is a partial function that maps a node or an edge, a property label, and a time point to a value of the property at that time point.

A valid TGraph conceptually corresponds to a sequence of valid conventional (non-temporal) graphs. This imposes the following conditions: (i) a condition on ξ^T that an edge can only exist at a time when both of the nodes it connects exist; and (ii) a condition on λ^T that a property can only take on a value at a time when the corresponding node or edge exists. Finally, we require that the property set of an entity **not** be empty at any time point when it exists. Practically, we require that each node and edge assign a value to a property called `type`.

Definition 2.1 associates graph nodes, edges and attribute values with *time points*. In the remainder of this paper, we will represent temporally adjacent time points by *intervals*, for syntactic compactness, as illustrated in Figure 1. Following the SQL:2011 standard, we use closed-open intervals, representing a discrete contiguous set of time points from Ω^T. This representation does not add expressiveness to a point-based representation, and is purely a syntactic device [10].

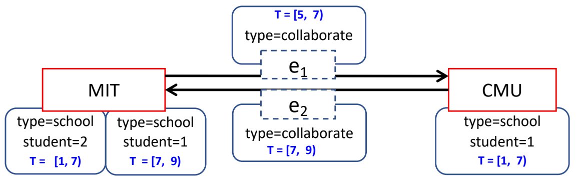


Figure 2: Result of aZoom^T over $\mathcal{G}_1$ (Figure 1). Semantically, this operation is executed over every snapshot of $\mathcal{G}_1$ to: (i) create `school` nodes for each value of the school property of `person` nodes in $\mathcal{G}_1$; (ii) count the number of persons enrolled at a school, set the value of the student property of the `school` node to that count; (iii) create edges of type `collaborate` between school nodes for which `co-author` edges were present in $\mathcal{G}_1$; and (iv) temporally coalesce the result across snapshots, due to point semantics.

We now describe aZoom^T and wZoom^T in detail using our running example, and refer to [37] for a formal treatment.

2.2 Attribute-Based Zoom

Temporal attribute-based zoom, denoted aZoom^T, is a temporal generalization of the graph node creation operation [42]. Node creation over non-temporal graphs takes a graph pattern as input, and computes a new node for each occurrence of a match of the pattern in the input. To assign identity to new nodes, it is customary to extend this operation with a Skolem function f_s. aZoom^T will similarly create nodes in the output TGraph from disjoint groups of nodes in the input, such that nodes within a group agree on the values of all grouping attributes.

Conceptually, aZoom^T is executed over every snapshot of the input TGraph, and new nodes are assigned identity by a Skolem function f_s, which generates consistent assignments across time. In addition to creating new nodes, aZoom^T will also optionally compute values of node attributes using the aggregation function f_{agg}, including count, sum, min, max, average, and user-specified functions that are required to be commutative and associative. Next, aZoom^T computes edges as follows. Suppose that input nodes n and n' corresponds to output nodes g and g', respectively, and that edge e connects n to n'. Then, the output will contain the edge e, with g as its source and g' as its target. Essentially, the input edge is re-created in the output, and re-pointed.

Node creation, computation of node attribute values, and re-pointing of the edges, is executed over each snapshot of the input TGraph, under point semantics. As the final step, the result is then coalesced, associating a time interval of maximal length during which no change occurred with every newly-computed node and edge. We now illustrate aZoom^T with an example.

Example 2.2. Node *Ann* in Figure 1 is associated with a closed-open interval $T = [1, 7)$, signifying that the node existed in the graph for six consecutive time points with no change. *Bob* exists in the graph during $T = [2, 9)$, but with a change to its attributes at time 5, when `school=CMU` was added. School names are represented as values of the school property of person nodes.

We invoke aZoom^T to compute from $\mathcal{G}_1$ a TGraph where schools become nodes rather than values, as shown in Figure 2

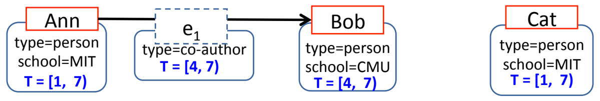


Figure 3: wZoom^T($\mathcal{G}_1$, window=3-months, nodes=all, edges=all, node.school=last(school)) over $\mathcal{G}_1$ (Figure 1).

with school nodes *MIT* and *CMU*. Note that the number of students at MIT changes over time: both *Ann* and *Cat* study there during $T = [1, 7)$, while only *Cat* studies there during $T = [7, 9)$. Note that both edge e_1 and e_2 have been redirected to newly created nodes and their validity period is updated to correct values based on when those were valid in the graph: While e_1 is valid during $T = [2, 7)$ in Figure 1, it is only valid during $T = [5, 7)$ in Figure 2, because *Bob* was not at CMU during $T = [2, 5)$.

2.3 Temporal Window-Based Zoom

The wZoom^T operator is analogous to *moving window temporal aggregation* in temporal relational algebra. This operator is inspired by stream aggregation of Li et al. [29] (adopted to graphs), and by generalized quantifiers [22].

The wZoom^T operator modifies validity periods of TGraph nodes and edges, by mapping different states of a node or an edge to a single representative state. This mapping is based on a specification of a temporal window, such as 2 months or 10 years. If the specified window is finer than the temporal resolution of the input TGraph, the operation has no effect. For example, applying wZoom^T with 1-month windows to a TGraph in which evolution is recorded across years will simply return the input TGraph. Note that, because wZoom^T is required to produce a valid TGraph as output, this operation does not support overlapping windows.

Window specification is of the form n $\{unit|\text{changes}\}$, where n is an integer, and *unit* is a time unit (e.g., 10 min, 3 years). Window specification generates a temporal relation W with the schema $(d \mid T)$, where each tuple associates a window number d with its period of validity T. We additionally require node and edge existence quantifiers {all|most|at least n|exists}, where n is a decimal representing the percentage of the time during which a node or an edge existed, relative to the duration of the window. Quantifiers are useful for observing different kinds of temporal evolution. For example, to observe strong connections over a volatile evolving graph we may include nodes that span the entire window (nodes=all), and edges that span a large portion of the window (edges=most). We refer to all and exists as universal and existential quantification, respectively.

A related point is that a given node or edge should exist at most once at any given time point, and so we must specify how conflicts in attribute values are resolved by wZoom^T. The answer to this question is determined by the window aggregation functions, which specify, for each attribute of a node or an edge, which of its values to accept as a representative for the given temporal window. We support the window aggregation functions first, last, and any (the default).

Example 2.3. Consider again TGraph $\mathcal{G}_1$ in Figure 1, and suppose that time points represent the months of 2018, and are divided into fiscal year quarters as follows: window $W1$: time points 1, 2, 3; $T = [1, 4)$, window $W2$: time points 4, 5, 6; $T = [4, 7)$, window $W3$: time points 7, 8, 9; $T = [7, 10)$. How might we quantify the state of $\mathcal{G}_1$ during each quarter, a 3-month temporal window?

Figure 3 shows the temporally coalesced results of zooming out to quarters over $\mathcal{G}_1$ with nodes=all and edges=all.

Ann is present in windows $W1$ and $W2$ in the input in Figure 1, and so is associated with $T = [1, 7)$ in the result for both universal and existential quantification. In contrast, *Bob* is present in the input for all of $W2$ but for only part of $W1$, and so is returned with $T = [4, 7)$ in the result for nodes=all, and with $T = [1, 7)$ for nodes=exists. Finally, *Cat* is present for all of $W1$ and $W2$, but for only part of $W3$ in the input (it is missing at time point 9), and so is associated with $T = [1, 7)$ in the output undernodes=all and with $T = [1, 10)$ under nodes=exists. Quantification is applied to edges analogously: e_1 is mapped to window $W2$ and e_2 is absent in the output in Figure 3, because there does not exist a quarter during which e_2 exists continuously in the input.

3 EVOLVING GRAPH REPRESENTATIONS AND DATAFLOW OPERATORS

In this section, we introduce several physical representations for a TGraph and detail how to define the zoom operations according to these representations. We express the zoom operators using general dataflow operations — directed acyclic graphs of operators resembling parallelizable second-order functions that execute user-defined first order functions. This is a popular programming model for distributed computations supported by systems such as Apache Spark [43] and Apache Flink [2].

We use the term *snapshot* to refer to a conventional (non-temporal) graph that represents the state of a TGraph during some interval in which no change occurred. Figure 4 shows the TGraph in our running example as a sequence of snapshots. When storing and accessing evolving graphs, we are concerned with preserving two kinds of locality: temporal and structural. Adopting the terminology of [19], with *structural locality*, neighboring vertices (resp. edges) of the same snapshot are laid out together, while with *temporal locality*, consecutive states of the same vertex (resp. edge) are laid out together. We develop four TGraph representations that differ in compactness and in the kind of locality (structural or temporal) they prioritize.

Representative Graphs (RG). RG represents a TGraph by a sequence of snapshots (conventional graphs), associating them with time intervals, see Figure 4 for an example. The snapshot sequence is by far the most common representation in the literature [15, 20, 24–26, 38, 40]. RG has the following schema:

```
TemporalGraph { interval: Interval,
                          snapshots: array(Snapshot) }
Snapshot { vertices: array(Vertex), edges: array(Edge) }
Interval { start: Date, end: Date }
Vertex { vid: long, type: string, attributes: dictionary }
Edge { eid: long, type: string, v1: Vertex, v2: Vertex,
      attributes: dictionary }
```

Note that vertices and edges of each snapshot store the attribute values for the interval represented by the snapshot. This representation is simple, and lends itself well to parallelizing operations in a distributed environment, as we can simply assign different snapshots to different workers. An advantage of RG is that it naturally preserves structural locality, and so is efficient for snapshot-based operations. An important drawback of RG is that it is not compact: in many real-world evolving graphs there is an 80% or larger overlap between consecutive snapshots [8].

Vertex Edge (VE). As illustrated in Figure 5, VE is a nested temporal relational representation of TGraph, with one relation storing vertices and the other edges, together with the corresponding

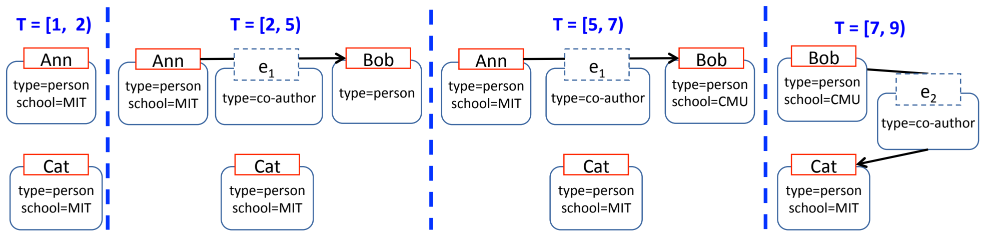


Figure 4: Representative-Graphs (RG): a "sequence of snapshots" representation of the TGraph $\mathcal{G}_1$ of Figure 1.

Vertices (V)

v	a	T
Ann	type=person, school=MIT	[1, 7)
Bob	type=person	[2, 5)
Bob	type=person, school=CMU	[5, 9)
Cat	type=person, school=MIT	[1, 9)

Edges (E)

e	v1	v2	a	T
e1	Ann	Bob	type=co-author	[2, 7)
e2	Bob	Cat	type=co-author	[7, 9)

Figure 5: Vertex-Edge (VE): nested relational representation of the TGraph $\mathcal{G}_1$ from Figure 1. The relations Vertices (V) and Edges (E) are temporally coalesced.

Vertices (V)

v	a
Ann	{ T=[1,7): type=person, school=MIT }
Bob	{ T=[2,5):type=person, T=[5,9):type=person, school=CMU }
Cat	{ T=[1,9): type=person, school=MIT }

Edges (E)

e	v1	v2	a
e1	Ann	Bob	{ T=[2,7): type=co-author }
e2	Bob	Cat	{ T=[7,9): type=co-author }

Figure 6: One Graph (OG): nested relational representation of the TGraph $\mathcal{G}_1$ from Figure 1. The relations Vertices (V) and Edges (E) are temporally coalesced.

time intervals. Both relations are temporally coalesced, giving rise to a compact representation. VE stores all vertex properties together as a single nested attribute (and all edge properties analogously). VE has the following schema:

```
TemporalGraph { interval: Interval,
    vertices: array(Vertex),  edges: array(Edge) }
Interval { start: Date, end: Date }
Vertex { vid: long, type: string, interval: Interval,
        attributes: dictionary }
Edge { eid: long, type: string, vid1:long, vid2: long,
      interval: Interval, attributes: dictionary }
```

For edges, we store a unique edge identifier **eid**(long) to support multi-graphs, as well as the vertex identifiers **vid1**(long) and **vid2**(long) that are foreign keys referring to the vertex relation. The main advantage of VE's attribute representation is that it lends itself to schema evolution. A disadvantage is that different properties may have different evolution rates, and a change to a single property requires a new vertex or edge tuple. VE stores graph vertices and edges in unordered collections, and therefore does not maintain temporal locality by default in cases where the state of a vertex or edge changes. For example, two tuples for vertex *Bob* in Figure 5 may not be located consecutively, or even on the same worker, once the data is partitioned across a cluster. We can reconstruct temporal locality at runtime, by re-partitioning the data based on vertex or edge identifiers.

One Graph (OG), One Graph Columnar (OGC). These are two topologically compact representations. OG stores all vertices and edges once, in a single aggregated data structure, as shown in Figure 6. In OG, vertices and edges store the history of the evolution of their attributes as an array of key-value pairs, together with the corresponding validity periods. Figure 6 shows the OG representation for our example graph. Note that we have only one tuple for vertex *Bob*, which holds two sets of values for two corresponding validity periods T=[2,5) and T=[5,9). OG has the following schema:

```
TemporalGraph { interval: Interval,
    vertices: array(Vertex), edges: array(Edge) }
Interval { start: Date, end: Date }
HistoryItem { interval: Interval, attributes: dictionary }
Vertex { vid: long, type: string,
                        history: array(HistoryItem) }
Edge { eid: long, type: string, v1: Vertex, v2: Vertex,
      history: array(HistoryItem) }
```

The schemas for OG and VE are similar in many ways. The main difference is that the interval and attribute dictionary in VE has been replaced with a **history** array that contains **HistoryItem**s. Each such history item stores an **interval** as the key and a dictionary of the corresponding **attributes**. The second difference is that OG contains a copy of the source and target vertex of each edge, instead of a foreign key to the vertex relation.

OGC, on the other hand, only stores the graph topology with validity periods as a graph, as shown in Figure 7. OGC has the following schema:

```
TemporalGraph { intervals: array(Interval),
         vertices: array(Vertex), edges: array(Edge) }
Interval { start: Date, end: Date }
Vertex { vid: long, type: string, intervals: Bitset }
Edge { eid: long, type: string, v1: Vertex,
      v2: Vertex, intervals: Bitset }
```

OGC is intended for topology-only attribute-less graphs, encoding the presence of a vertex or edge in each interval with a bitset. Both OG and OGC emphasize temporal locality, while also preserving structural locality, but lead to a much denser graphs than RG. This, in turn, makes parallelizing computation challenging.

In the remainder of this section, we describe how to define aZoom^T and wZoom^T in terms of dataflow operations according to our proposed representations.

3.1 Attribute-Based Zoom

We now describe aZoom^T for each TGraph representation. In the algorithms we present, V and E are overloaded to refer to the vertex and edge relations of a given snapshot (in the case of RG) or of the overall TGraph. In aZoom^T we use a Skolem function f_s to produce new vertex ids. f_s is a user-provided function that

Bitset (b): T={[1,2),[2,7),[7,9)}

Vertices (V)

v	b
Ann	[1, 1, 0]
Bob	[0, 1, 1]
Cat	[1, 1, 1]

Edges (E)

e	v1	v2	b
e1	Ann	Bob	[0, 1, 0]
e2	Bob	Cat	[0, 0, 1]

Figure 7: One Graph Column (OGC): nested relational representation of the TGraph $\mathcal{G}_1$ of Figure 1. Vertices (V) and Edges (E) are temporally coalesced. Bitsets represent validity during periods of T={[1,2),[2,7),[7,9)}.

takes the vertex id and all attributes as an input and produces a long identifier as output. We additionally apply the commutative and associative aggregation function f_{agg} to resolve cases where we have a series of vertices with identical identifiers but multiple values for the same attribute in the same snapshot. This is an important step that ensures that each snapshot in the result corresponds to a valid graph (see [36] for details).

RG. Recall thatRG maintains a collection of snapshots. We apply the same set of operations in an embarrassingly parallel manner to each snapshot, as there are no dependencies between them in this case (Algorithm 1). We iterate over each snapshot (lines 3-10) and return an RG (line 11) containing the aZoom^T result. We apply f_s to each vertex using a map (line 5) in order to compute a new identifier for each vertex. The copyWithVid function updates each vertex identifier while keeping other attributes unchanged. We then group vertices by id (line 7) and apply the aggregation function f_{agg} (line 8).

To redirect edges to the newly created vertices, we apply the function f_s to the vertices v1 and v2 of each edge in a map (line 9). The copyWithVids function updates the id of the vertices to the new identifiers. The edges contain a copy of their source and target vertices in RG, which obliviates the need for a join here.

Algorithm 1 aZoom^T over RG

```
Require: Skolem function $f_s : V \Rightarrow \mathbb{N}$; Aggregation function $f_{agg}$ :
    $V \times V \Rightarrow V$
 1: $newSnapshots \leftarrow \emptyset$
 2:                                          ▷ Aggregate each snapshot
 3: for ($V$,E) in graph.snapshots do
 4:     $V' \leftarrow V$                          ▷ Update of vertex identifiers
 5:         .map{$v \Rightarrow v$.copyWithVid($f_s(v)$)}
 6:                          ▷ Vertex aggregation for identity-equivalence
 7:         .groupBy{$v \Rightarrow v.vid$}
 8:         .reduce{$(v_a, v_b) \Rightarrow f_{agg}(v_a, v_b)$}
                                     ▷ Edge redirection to new vertices
 9:     $E' \leftarrow E$.map{$e \Rightarrow e$.copyWithVids($f_s(e.v1), f_s(e.v2)$)}
10:     Add ($V'$, $E'$) to $newSnaphots$
11: return new TGraph $G(newSnapshots)$
```

VE. VE consists of two temporal relational tables for vertices and edges, which contain tuples for each vertex or edge history. Algorithm 2 details our implementation of aZoom^T for VE. We first calculate non-overlapping intervals (lines 2-5) based on the temporal splitter concept introduced in [11]. We join intervals and vertices (lines 7- 9), assign new identifiers (line 10), and enforce identity-equivalence in each interval with the aggregation function (line 12). Since VE edges only contain a foreign key to the corresponding vertices, we need to join the edges with their corresponding vertices for the edge redirection process (lines 14 and 15), before we can apply the f_s function to each corresponding vertex to redirect the edge (line 18).

Algorithm 2 aZoom^T over VE

```
Require: Skolem function $f_s : V \Rightarrow \mathbb{N}$; Aggregation function $f_{agg}$ :
    $V \times V \Rightarrow V$
 1: $I \leftarrow V$      ▷ Non-overlapping intervals for each new vertex identifier
 2:     .map{$v \Rightarrow (f_s(v), v.interval)$}
 3:     .groupBy{$(vid, \_) \Rightarrow vid$}
 4:     .foldLeft(EmptyInterval)
 5:         {$(i, v) \Rightarrow$ mergeNonOverlapping($i, v.interval$)}
 6: $V' \leftarrow V$            ▷ Vertex aggregation for non-overlapping intervals
 7:     .join($I$).on{$(v, id) \Rightarrow v.id == i.vid$}
 8:     .flatMap{$(v, i) \Rightarrow$ verticesForIntervals($v, i$)}
 9:     .map{$(v, i) \Rightarrow$
10:         $v$.copyWithIdAndInterval($f_s(v), i$) }
11:     .groupBy{$v \Rightarrow v.id$}
12:     .reduce{$(v_a, v_b) \Rightarrow f_{agg}(v_a, v_b)$}
13: $E' \leftarrow E$                          ▷ Edge redirection to new vertices
14:     .join($V$).on{$(e, v) \Rightarrow e.vid1 == v.id$}
15:     .join($V$).on{$((e, \_), v) \Rightarrow e.vid2 == v.id$}
16:     .map{$(e, v1, v2) \Rightarrow$
17:         $i \leftarrow$ recomputeInterval($e, v1, v2$)
18:         $e$.copyWithVidsAndInterval($f_s(v1), f_s(v2), i$) }
    return new TGraph $G(V', E')$
```

OG. We implement aZoom^T for One Graph (OG) analogously to RG, with the difference that we compute over the entire TGraph rather than over each individual snapshot (Algorithm 3). We split each vertex in OG based on its history, and apply the f_s function to each element of the history array individually. We use a flatMap function on vertices combined with a map on the history elements of each vertex for this (lines 1-3). We again enforce identity-equivalence with our aggregation function (lines 4 and 5). The vertext computation portion of Algorithm 3 is illustrated in Figure fig:az-og. For edge redirection in OG, we split the edges by expanding the history of each corresponding vertex in that edge, as OG stores each edge only once. Next, we apply the Skolem function f_s to each element of the history (line 6-9).

Algorithm 3 aZoom^T over OG

```
Require: Skolem function $f_s : V \Rightarrow \mathbb{N}$; Aggregation function $f_{agg}$ :
    $V \times V \Rightarrow V$
 1: $V' \leftarrow V$ .flatMap{$v \Rightarrow$
 2:     $v.history$.map{$(\_, attr) \Rightarrow$
 3:         $v$.copyWithIdAndAttributes($f_s(v.vid), attr$) }}
 4: .groupBy{$v \Rightarrow v.vid$}
 5: .reduce{$(v_a, v_b) => f_{agg}(v_a, v_b)$}
 6: $E' \leftarrow E$ .map{$e \Rightarrow$
 7:     $h \leftarrow$ recompute_history($e$)
 8:     $e$.copyWithVidsAndHistory($f_s(e.v1.vid)$,
 9:         $f_s(e.v2.vid), h$)}
    return new TGraph $G(V', E')$
```

OGC does not represent attributes and so does not support aZoom^T.

3.2 Temporal Window-Based Zoom

As we did for aZoom^T, we express wZoom^T differently for each representation, with some common aspects. The first step is to

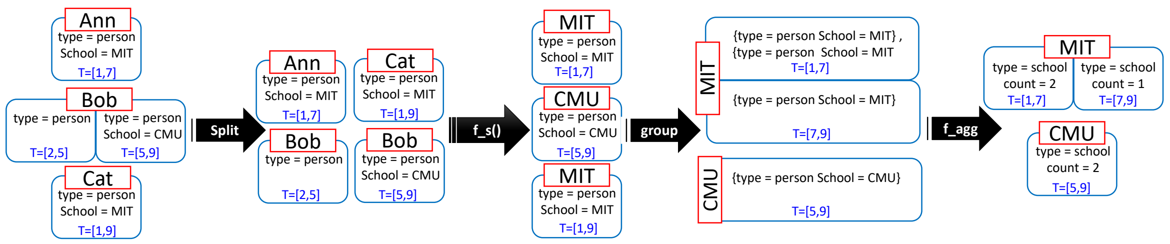


Figure 8: Illustration of the vertex computation portion of Algorithm 3, aZoom^T, over TGraph in Figure 6, with count as f_{agg}. The first two steps correspond to the call to flatMap on lines 1-3: splitting nodes based on their history array, and then calling the Skolem function f_s to generate ids for new nodes. In this example, f_s outputs the value of the school property. The next step groups vertices by id (line 4). The final step (line 5) applies the aggregation function count, storing the computed value as a vertex property.

compute the temporal window relation based on the window specification. We split the total graph lifetime temporally by applying the function computeNewIntervals to the graph. This function takes an interval as an input and returns a tuple containing the old and the recomputed interval.

A major difference to aZoom^T is that the TGraph must be coalesced *before* wZoom^T can be applied, in order to guarantee the correctness of the zoom operation. This is because aZoom^T executes over each snapshot (under snapshot reducibility), while the computation of wZoom^T is across snapshots. Consequently, if the input to wZoom^T is not coalesced, we cannot properly apply existence quantifiers and compute results of aggregation.

We additionally need to handle potential dangling edges for all representations in wZoom^T to ensure that every snapshot of the resulting TGraph is a valid graph, as specified in the condition over ξ^T in Definition 2.1. Recall that wZoom^T supports the quantifiers all, most, at least n, and exists, which can be translated to a threshold on the percentage of the time during which an entity (a vertex or an edge) existed, relative to the duration of the window: $t = 1$ for all, $t > 0.5$ for most, $t > 0$ for exists and $t > n$ for at least n. If an entity's existence meets the threshold, it will be retained in the result of the operation. A dangling edge check is only required if r_v is more restrictive than r_e, because a particular edges may pass the check, but one or more of the vertices it connects may not.

RG implements wZoom^T as shown in Algorithm 4. We again use the computeNewInteval function to compute the new intervals based on the window specification (line 2). Next, we apply join, groupBy, and flatMap to map each vertex to one or more snapshots from the specification (lines 4-9). Then, vertices are grouped by their id within each new interval (line 10). Next, we filter vertices and edges based on the existence quantifier (line 11). We apply the math_threshold function to vertices with their respective thresholds (r) to filter vertices that do not meet the criteria of our quantifier. Finally, we apply the resolve function f_v to compute the new attribute values (line 12). We treat edges analogously (lines 14-18). At the end, we merge snapshots into a TGraph and remove dangling edges.

VE implements wZoom^T using Algorithm 5. Figure 9 illustrates this algorithm for vertex *Bob* from Figure 5. We first need to calculate the new intervals using computeNewInterval (lines 2-3). Then we join V with the intervals to align each vertex with each temporal window (lines 4-6) to split the vertices. Next, we group by interval and vertex (line 7), and filter vertices that do not pass the quantifier threshold (line 8). Finally, we resolve the vertices' final attributes (line 12). We apply the same operations

Algorithm 4 wZoom^T over RG

```
Require: resolve functions $f_v, f_e$; quantifiers $r_v, r_e$
 1:                                        ▷ Computation of new intervals
 2: $I' \leftarrow I.\mathbf{map}\{i \Rightarrow (i, \text{computeNewInterval}(i))\}$
 3:                                        ▷ Grouping of snapshots by new interval
 4: $S \leftarrow G.snapshots.\mathbf{join}(I')$
 5:     $.\mathbf{on}\{(s, interval) \Rightarrow s.i == interval.i\}$
 6:     $.\mathbf{groupBy}\{(s, interval) \Rightarrow interval.newInterval\}$
 7:                                        ▷ Aggregation of vertices for new snapshots
 8: $V' \leftarrow S.\mathbf{flatMap}\{(i, snapshot) \Rightarrow$
 9:     $(i, snapshots.\text{map}\{s \Rightarrow s.vertices\})\}$
10:     $.\mathbf{groupBy}\{(i, v) \Rightarrow (i, v.id)\}$
11:     $.\mathbf{filter}\{(i, vertices) \Rightarrow \text{match\_threshold}(vertices, r_v)\}$
12:     $.\mathbf{reduceByKey}\{((v_a), (v_b)) \Rightarrow f_v(v_a, v_b)\}$
13:                                        ▷ Aggregate edges for new snapshots
14: $E' \leftarrow S.\mathbf{flatMap}\{(i, snapshot) \Rightarrow$
15:     $(i, snapshots.map\{s \Rightarrow s.edges\})\}$
16:     $.\mathbf{groupBy}\{(i, e) \Rightarrow (i, e.id)\}$
17:     $.\mathbf{filter}\{(i, edges) \Rightarrow \text{match\_threshold}(edges, r_e)\}$
18:     $.\mathbf{reduceByKey}\{((e_a), (e_b)) \Rightarrow f_e(e_a, e_b)\}$
                                           ▷ Recreate RG representation
19: $G' \leftarrow \text{merge}(I', V', E')$
```

to edges (lines 11-18). We remove dangling edges (given that $r_v > r_e$) with two semijoins (lines 17-19).

OG implements wZoom^T using Algorithm 6. Recall that in OG each vertex stores its interval information in a history array. We process each element of this array separately and rebuild the array afterwards (lines 1-4) for this process. We first invoke recomputeIntervals (line 2) to recompute the history array with updated intervals. Next, we leverage the aggregateAndFilterAttributes function (line 3) to group, filter and resolve vertices analogous to previous algorithms, and apply the same transformations to the edges as well (lines 5-8).

We again remove dangling edges with semijoins (lines 9-15). The only difference here is that joining edges with vertices is not enough, as we also need to update the history arrays. We achieve this with a map function which updates every edge history with the intersection of the edge history and the corresponding vertex history (lines 12 and 15) using the copyWithHistory function.

OGC implements wZoom^T similarly to OGC, but working with a bitset instead of a history array. Removing dangling edges in OGC is as simple as computing the logical and between the edge bitset and the corresponding vertex bitsets.

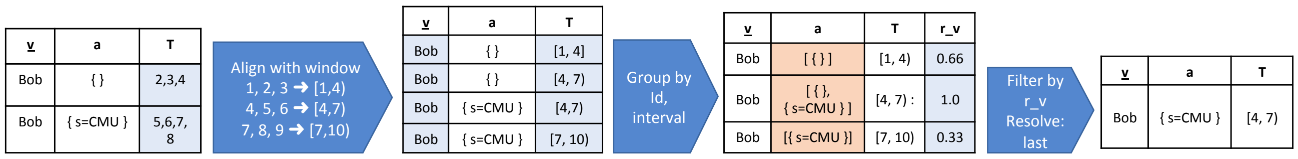

v | a | T
Bob | { } | 2,3,4
Bob | { s=CMU } | 5,6,7, 8
Align with window 1, 2, 3 ➔ [1,4) 4, 5, 6 ➔ [4,7) 7, 8, 9 ➔ [7,10)
v | a | T
Bob | { } | [1, 4]
Bob | { } | [4, 7)
Bob | { s=CMU } | [4,7)
Bob | { s=CMU } | [7, 10)
Group by Id, interval
v | a | T | r_v
Bob | [{ }] | [1, 4) | 0.66
Bob | [{ }, { s=CMU }] | [4, 7) : | 1.0
Bob | [{ s=CMU }] | [7, 10) | 0.33
Filter by r_v Resolve: last
v | a | T
Bob | { s=CMU } | [4, 7)

Figure 9: Illustration of Algorithm 5, wZoom^T, for vertex *Bob* in Figure 5, with window size 3 and last as f_v. The first step aligns each vertex with each temporal window (lines 4-6 of the algorithm). Next we create a single nested representation of each vertex per window and compute r_v, the fraction of the window during which the vertex was observed (line 7). Finally, we filter vertices by r_v and resolve their attribute values with f_v =last (lines 8, 9).

Algorithm 5 wZoom^T over VE

```
Require: resolve functions $f_v, f_e$; quantifiers $r_v, r_e$
 1:                                              ▷ Computation of new intervals
 2: $I' \leftarrow I$.map{ $i \Rightarrow (i,$ computeNewInterval($i$)) }
 3:                                              ▷ Vertex aggregation for new intervals
 4: $V' \leftarrow V$.join($I'$).on{ $(v, (i, n)) \Rightarrow v.n == i$ }
 5:     .map { $(v, (i, newInterval)) \Rightarrow$
 6:         $v$.copyWithNewInterval($newInterval$)}
 7:     .groupBy{ $v \Rightarrow (v.id, v.interval)$ }
 8:     .filter{$(i, vertices) \Rightarrow$ match_threshold($vertices, r_v$)}
 9:     .reduceByKey{$((v_a), (v_b)) \Rightarrow f_v(v_a, v_b)$}
10:                                              ▷ Edge aggregation for new intervals
11: $E' \leftarrow E$.join($I'$).on{ $(e, (i, n)) \Rightarrow e.interval == n$ }
12:     .map { $(e, (i, newInterval)) \Rightarrow$
13:         $e$.copyWithNewInterval($newInterval$)}
14:     .groupBy{ $e \Rightarrow (e.id, e.interval)$ }
15:     .filter{$(i, edges) \Rightarrow$ match_threshold($edges, r_e$)}
16:     .reduceByKey{$((e_a), (e_b)) \Rightarrow f_e(e_a, e_b)$}
17:     if $r_v > r_e$ then                      ▷ Dangling edge removal
18:         $E'' \leftarrow E'$.semijoin($V'$)
             .on{ $(e, v) \Rightarrow e.vid1 == v.id$ and in_interval(e, v)}
19:         $E''' \leftarrow E''$.semijoin($V'$)
             .on{ $(e, v) \Rightarrow e.vid2 == v.id$ and in_interval(e, v)}
20: return new TGraph ($V', E'''$)
```

Algorithm 6 wZoom^T over OG

```
Require: resolve functions $f_v, f_e$; quantifiers $r_v, r_e$
 1: $V' \leftarrow V$.map{$v \Rightarrow$
 2:     $h \leftarrow$ recomputeIntervals($v.history$)
 3:     $h \leftarrow$ aggregateAndFilterAttributes($h, f_v, r_v$)
 4:     $v$.copyWithHistory($h$) }
 5: $E' \leftarrow E$.map{$e \Rightarrow$
 6:     $h \leftarrow$ recomputeIntervals($e.history$)
 7:     $h \leftarrow$ aggregateAndFilterAttributes($h, f_e, r_e$)
 8:     $e$.copyWithHistory($h$) }
 9: if $r_v > r_e$ then                          ▷ Dangling edge removal
10:     $E'' \leftarrow E'$.semijoin($V'$)
         .on{ $(e, v) \Rightarrow e.vid1 == v.id$ and in_interval(e, v)}
11:     .map{$(e, v) \Rightarrow$
12:         $e$.copyWithHistory(intersect($e.history, v.history$)) }
13:     $E''' \leftarrow E''$.semijoin($V'$)
         .on{ $(e, v) \Rightarrow e.vid2 == v.id$ and in_interval(e, v)}
14:     .map{$(e, v) \Rightarrow$
15:         $e$.copyWithHistory(intersect($e.history, v.history$)) }
16: return new TGraph $G(V', E'')$
```

4 IMPLEMENTATION

We defined our zoom operators in Section 3 using general dataflow operations and UDFs that are implemented by a variety of popular systems. Apache Spark with GraphX [17] and Apache Flink with Gelly [7] are natural candidates for such workloads, as is Differential Dataflow [33]. We choose Apache Spark for our implementation due to its maturity and popularity.

Our implementation includes a TGraph API, several graph representations as discussed in Section 3, and several optimizations such as lazy coalescing. Our API supports chaining multiple operations together and switching between graph representations during query execution.

The VE representation is implemented directly over Spark's Resilient Distributed Datasets (RDDs) [43] while RG, OG and OGC leverage the GraphX library for static graphs [17]. We use the long datatype to represent node and edge identifiers to maintain interoperability with GraphX.

GraphX-specific implementation details. GraphX implements vertex-cut-based partitioning that reduces communication overhead [17] for certain aggregations on graphs. GraphX also provides an optimized implementation of a distributed triplet view, a concept originating from Resource Description Frameworks (RDF) [31]. The triplet view provides fast access to each edge and its corresponding source and destination vertex properties. The triplet view requires a materialized three-way join, which GraphX optimizes by implementing vertex-mirroring and a multicast join [17]. We leverage the implementation of the triplet view to efficiently access edges' vertex attributes in RG, OG and OGC. We implement RG as sequence of GraphX graphs, while OG and OGC are modeled as a single GraphX graph. GraphX mechanisms such as vertex-cut partitioning and the triplet view enabled us to implement graph operations more efficiently.

Data loading. The data is read from the Hadoop Distributed File System (HDFS). Our on-disk data layout uses Apache Parquet, a columnar data format for HDFS based on the Dremel project [34]. Apache Parquet does not have a mechanism for indexing, but it supports filter pushdown on any column by which the data is sorted on disk. We store and load vertices and edges as separate vertex and edge Parquet files. The default schema to store a graph on disk is similar to the VE schema described in Section 3. We load two of our representations (VE and RG) from this format. To apply a filter pushdown, the data on disk need to be sorted. For VE, we use the vertex/edge identifier as the first sort key, and the interval start time as the second key. Storing data in this way preserves temporal locality, and places the history of changes in a vertex or an edge together. Parquet does not support filter pushdown for datetime formats, hence we store time as UNIX timestamps (long).

We use a similar schema for RG, however, we sort vertices and edges by the interval start time first, and then by their vertex (resp. edge) identifier, to preserve structural locality. During our experiments we learned that RG can be loaded 30% faster using the structural locality instead of temporal locality (experiment omitted due to space constraints). While OG and OGC could be loaded in the same way as VE, we experimentally validated that it is significantly faster to pre-compute nested versions of the graphs with schemas described in Section 3, and then convert to OG or OGC during load time. A problem with this approach is that Parquet's filter pushdown will not work, since interval information is stored in a nested column. We resolve this issue by storing the first and last time a vertex/edge existed as a separate column on disk, and sorting on these columns.

We provide a GraphLoader utility that can initialize any of our physical representations from Apache Parquet files on HDFS or on local disk. This loader accepts a date range and filters the data through Parquet's filter pushdown. For datasets with a long evolution history, this optimization provides a substantial performance improvement (see [36]).

Coalescing. The coalesce primitive for merges adjacent and overlapping time periods for value-equivalent tuples. Several implementations have been suggested for the coalesce operation over temporal SQL relations [5]. In our implementation for VE, we use the partitioning method: grouping the vertex and the edge relation by key, then sorting by start time, and folding tuples within each group and checking pairs of adjacent tuples for value-equivalence. The effect of this operation is that a single tuple is produced for each period of maximum length during which no change occurred.

To further optimize performance, we coalesce lazily for sequences of two or more operations. Recall that aZoom^T computes in each snapshot, and so it does not require its input to be temporally coalesced to produce the correct output. In contrast, wZoom^T does require its input to be coalesced for correctness, because it computes across snapshots. This means that, in a sequence of aZoom^T and wZoom^T operators, the system does not need to temporally coalesce before invoking aZoom^T, but it must coalesce before invoking wZoom^T and at the end of the operator sequence, when the final result is produced.

5 EXPERIMENTAL EVALUATION

We conduct an experimental evaluation to study the performance of aZoom^T and wZoom^T. Our goal is to understand how different representations and their corresponding operator implementations perform for different datasets and workloads. We present three different categories of experiments: aZoom^T experiments (Section 5.1), wZoom^T experiments (Section 5.2), and experiments combining both operations (Section 5.3).

Cluster. All experiments are conducted on a 16-worker in-house Cloudera cluster, using Linux CentOS 14.04 and Spark v2.2. Each machine has 4 cores and 32 GB of RAM. Spark standalone cluster manager and Hadoop 2.6 were used. In each experiment, we report the mean runtime of three executions, each with a cold start. The runtime includes the setup time of submitting a job to the cluster manager, reading the data from disk, executing the operation, and materializing the results in memory. We set a 30-minute time-out for all experiments.

Datasets. We evaluate the performance on two real world datasets, WikiTalk and NGrams, and a family of synthetic datasets SNB, with different scale factors. All datasets are summarized in the table below, and differ in size, in the number and type of attributes, and in evolution rates, calculated as the average graph edit similarity [38] between consecutive snapshots (the edit similarity between snapshots i and j is the ratio of the number of common edges to the sum of the number edges: $2 * |E_i \cap E_j|/(|E_i| + |E_j|)$). In contrast to WikiTalk and NGrams, SNB is a growth-only graph, and so it shows a higher evolution rate.

	vertices	edges	snaps	ev. rate
WikiTalk	2.9M	10.7M	179	14.4
SNB:10	65K	1.9M	36	89
SNB:100	448K	20M	36	90
SNB:300	1.1M	59M	36	90
SNB:1000	3.3M	202M	36	91
NGrams:M	28M	606M	287	16.6
NGrams:L	48M	1.32B	328	18.2

WikiTalk is a real dataset that contains over 10 million messaging events (edges) among 3 million wiki-en users (vertices) at a 1-month resolution, from 2000 through 2016 [41]. Vertices have two attributes: `name` is a unique username for each account and `editCount` is the number of edits committed by the user (around 15K unique values). Edges have no attributes. WikiTalk is a very sparse dataset with short-lived edges and growth-only vertices: once added, a vertex persists for the lifetime of the graph and its attributes do not change.

NGrams is a real dataset that contains word co-occurrence pairs, with 88 million word vertices (3.2 unique words in all) and over 2.8 billion undirected co-occurrence edges. In our experiments we use two versions of this dataset: NGrams:L, with 328 yearly snapshots from 1520 through 1920, and NGrams:M, with 287 yearly snapshots from 1520 through 1870. NGrams is denser than WikiTalk; its vertices persist over time, while edges can appear and disappear. This dataset exhibits a linear relationship between the number of nodes and the number of edges.

The LDBC Social Network Benchmark (SNB) [12] is a synthetic graph generator that produces realistic networks with different types of entities and different attributes. We focus on SNB person entities (vertices) and on friendship relationships (edges), and generate datasets at four scale factors: SNB:10, SNB:100, SNB:300, and SNB:1000, with 36 monthly snapshots in each. SNB:1000 is the largest dataset that can be created without changing the generator source code. SNB does not generate a temporal benchmark but, since entities and relationships have timestamps, it can be viewed as a growth-only evolving graph. We use the vertices attribute `firstName` (5300 unique values in SNB:1000), edges have no attributes. The friendship network generated using SNB is growth-only graph, a graph where every vertex and edge is added once and never goes away.

5.1 Evaluation of aZoom^T

We now evaluate the performance of attribute-based zoom. Experiments are executed with RG, VE and OG, as described in Section 2, but not with OGC, which does not support aZoom^T.

Fixed number of groups, varying data size. Dataset size plays an important role in the performance of aZoom^T. We simulated different data sizes by using three datasets and varying the number of snapshots in each dataset. In this experiment and all other experiment in this section, we used aZoom^T with a hash function as the Skolem function f_S that generate new ids based on one of the attributes of the graph. In WikiTalk we group by

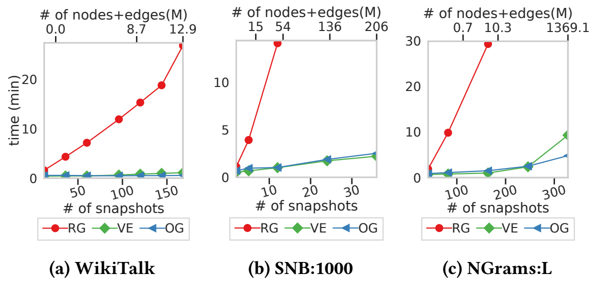


Figure 10: aZoom^T: The effect of dataset size on the runtime for each dataset. OG and VE perform on par, while RG quickly times out.

username (2.9M output groups), In NGrams— by word (3.2M output groups), and in SNB— by firstName (5,300 output groups).

Figure 10 shows the runtime of aZoom^T on different datasets. As expected, increasing the data size increases the execution time. OG is the best-performing representation, and VE is second-best. Both VE and OG exhibit sub-minute runtimes on WikiTalk: at most 0.54 min for OG and at most 1.09 min for VE (Figure 10a). The runtime of VE for SNB:1000 is at most 2.21 min for this graph with over 200M edges, where OG takes up to 2.53 minutes. (Figure 10b). Notably, OG scales well, even for NGrams, where OG computes in 4.8 min for 400 years worth of data and VE in 9.3, in a graph with 1.3 billion edges (Figure 10c). In contrast, RG is much slower than VE and OG, and it does not scale for the full SNB:1000 and NGrams:L dataset. It takes 26 minutes for WikiTalk, 14 minutes for 12 snapshots of SNB, timing out for anything larger and taking 29.55min to compute for 200 snapshots of NGrams, and timing out for 300 snapshots.

Fixed number of groups and graph size, varying number of snapshots. Another important factor in evolving property graphs that can impact operator performance is the number of snapshots (intervals during which no change occurred in the TGraph). We generate experimental datasets to measure this effect by merging consecutive snapshots of WikiTalk and NGrams:L, where we gradually decrease the number of intervals, while we keep the size of the dataset (in terms of the number of nodes and edges) fixed. For SNB:1000, we directly generate datasets with the desired number of snapshots. For WikiTalk, we select the last 160 months of history, and create graphs with between 2 and 160 snapshots. For NGrams, we select the last 320 years of the graph's history, and again generate datasets with between 2 and 320 snapshots. For SNB, we generate between 12 and 360 snapshots, corresponding to between 1 and 30 years worth of network evolution. Note that generating graphs in this way does neither change the number of nodes and edges, nor the group-by cardinality.

Figure 11 shows the runtime of aZoom^T as a function of the number of snapshots. OG and VE exhibit comparable performance for WikiTalk, executing in under 2 minutes, with OG being slightly more efficient. The trends are different in SNB: the runtime of aZoom^T on both OG and VE is near-constant, but VE is more efficient: 2.3 minutes for VE, compared to 2.9 minutes for OG. OG outperforms VE for NGrams; their runtime increases linearly with an increasing number of intervals.

The difference in performance across datasets is due to the nature of data evolution. WikiTalk and SNB only have one tuple per node, since attributes do not change over time, therefore an

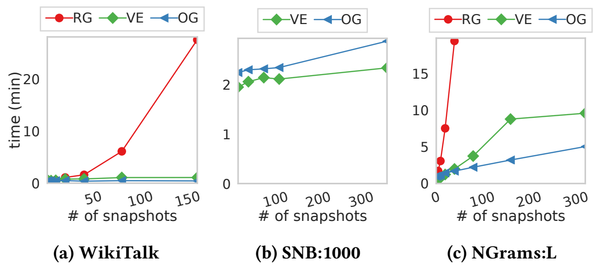


Figure 11: aZoom^T: Fixed dataset size and group-by cardinality, varying number of snapshots. The number of nodes and edges is fixed to the largest graph size, and the group-by cardinality is fixed to the natural group-by cardinality of each dataset.

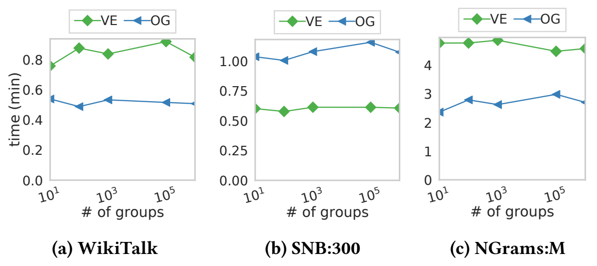
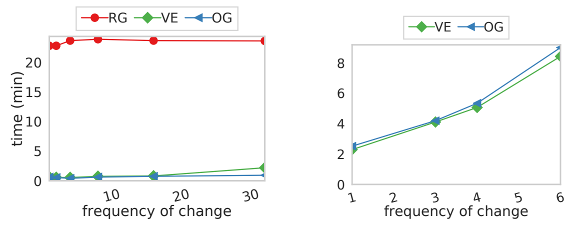


Figure 12: aZoom^T: Fixed dataset size and number of snapshots, with varying group-by cardinality.

increase in the number of intervals does not change the number of tuples (which is not the case for NGrams). We observe that RG is the least efficient representation for this operation, except for the smallest number of intervals in WikiTalk, where all representations have roughly similar performance. The running time of RG grows linearly with the number of intervals, with a high slope. We timed out this experiment at 30 minutes per execution, and RG failed to complete for SNB and NGrams at 80 intervals.

Fixed size and number of snapshots, varying group-by cardinality. In this aZoom^T experiment, we investigate the effect of group-by cardinality — the number of new nodes being created by the aZoom^T operation, on performance. We work with the WikiTalk, SNB:300 and NGrams:300 datasets at their original temporal resolution. We vary the number of groups in the output by assigning a group identifier to each node in the input. Group identifiers are drawn uniformly at random from a given integer range. We varied the range to control group-by cardinality, setting it to 10, 100, 1,000, 100,000, and 1,000,000. Figure 12 shows the results of this experiment. We observe that the runtime of aZoom^T over OG, VE and RG is not affected by group-by cardinality. For visibility purpose, we did not include RG in Figure 12. On WikiTalk, RG showed an execution time of about 29 minutes for all the group-by cardinality values.

Frequency of change. In our final aZoom^T experiment, we study the effect of the frequency of change on performance. Therefore, we synthetically change vertex attributes values with a fixed frequency. While this intervention does not change the size of the graph in terms of the number of nodes and edges, it does change the storage requirements (e.g., the number of tuples for VE, or the length of the array in OG) for each vertex.



(a) WikiTalk (b) SNB:1000

Figure 13: aZoom^T: Fixed dataset size and number of snapshots, varying frequency of change of vertex attributes

Figure 13 shows the effect of the frequency of change on the performance for WikiTalk (Figure 13a) and SNB:1000 (Figure 13b). The size of each graph and the contained number of snapshots is fixed to the full dataset size. While the group-by cardinality does vary, the number of new groups is of the same order of magnitude as in the original graphs. We observe that the frequency of change has no effect on the performance of RG. This is because RG stores each vertex once per snapshot, irrespective of whether there was a change between consecutive snapshots. The runtime of aZoom^T on OG is higher when more changes occur. This is expected: Recall that OG stores attributes with their corresponding validity intervals in an array, and so a higher frequency of change results in longer arrays, which slows down operations on OG. VE stores each change as a new tuple,and a higher frequency of change results in more tuples, slowing down VE as well.

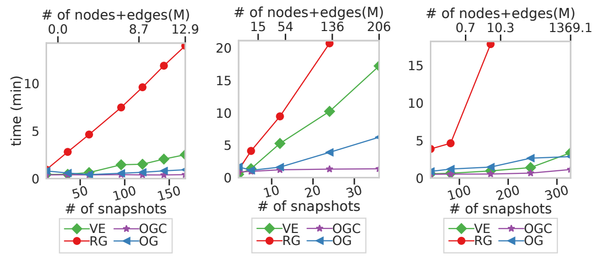
Summary. We studied the effects of data size, the number of snapshots, the frequency of attribute change, and the group-by cardinality (e.g., the number of newly-computed nodes) on the performance. We observed that OG is the best representation for aZoom^T, followed by VE. For our largest experimental datasets, with over 1.3 billion edges, aZoom^T can be executed in less than 5 minutes with OG. The dataset size (the total number of nodes and edges) affects operator performance on all datasets. The number of representative graphs (snapshots) has a small effect on VE and OG, and a significant effect on RG. We did not observe an effect of the group-by cardinality on the runtime in any representation. The frequency of change has a small effect on RG, but it affects VE and OG significantly.

5.2 Evaluation of wZoom^T

We now investigate the performance of wZoom^T. In all experiments, we load RG from disk enforcing structural locality, and VE enforcing temporal locality. For OG and OGC we load data from nested format described in Section 4.

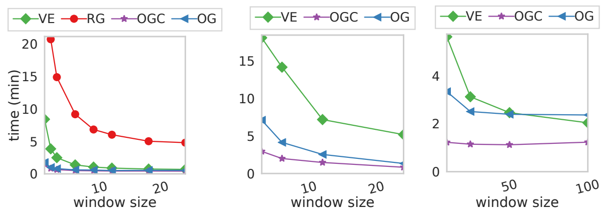
Fixed time window, changing data size. In this experiment, we fix the zoom window size to 3 snapshots for WikiTalk (grouping into up to 60 temporal windows) and SNB:1000 (grouping into up to 12 temporal windows), and 25 snapshots for NGrams:L (grouping into up to 16 temporal windows). We load different temporal slices of each graph and measure the execution time of wZoom^T. Figure 14 shows the results of this experiment. We applied “exists“ quantifiers for both nodes and edges. We observed similar results for “all“ quantifiers (except that they make wZoom^T slightly faster as fewer nodes and edges have to be kept in the result), which we omit for space reasons.

As expected, increasing the size of the graph increases the runtime on all representations. Our implementation based on



(a) WikiTalk (b) SNB:1000 (c) NGrams:L

Figure 14: wZoom^T: Fixed window size, changing data size, nodes=exists, edges=exists



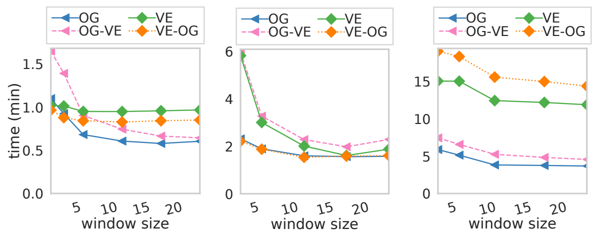
(a) WikiTalk (b) SNB:1000 (c) NGrams:L

Figure 15: wZoom^T: Fixed data size and number of intervals with varying window size, nodes=all, edges=all. OG and OGC outperform other representations.

OGC is the clear winner for all datasets, taking 0.41 minutes for WikiTalk, 1.25 minutes for SNB and 1.12 minutes for NGrams. For WikiTalk and SNB, OG is the second winner while VE performs better for NGrams:L, particularly for larger TGraph sizes. Finally, RG performs worst for all datasets. The reason for VE’s significant performance drop on SNB is window size. We look at the impact of window size on wZoom^T in the next section.

Fixed data size, varying temporal window size. In the previous experiment, we used a fixed zoom window size and increased the size of the graph. In this experiment, the size of the graph is fixed and we vary the size of the temporal window. Figure 15 shows the corresponding results. RG does not scale for temporal window-based zoom on large datasets, therefor we only report performance numbers for RG on WikiTalk. We observe that the performance of OG and OGC does not depend on the window size, while the operations on VE take longer to execute for smaller temporal windows. OGC is the winner among all datasets followed by OG; VE exhibits longer runtimes for smaller window sizes especially. This is because VE creates copies of each tuple in order to align them with the computed time windows. The smaller the window, the more tuples are created in the intermediate stage. This effect is more visible for WikiTalk and SNB because of their growth-only nature. In SNB, each vertex or edge exists from its start date to the life time of the graph, therefore VE needs to create a large amount of copies as each of those long intervals is split into intervals corresponding to the window size.

Summary. We studied the effect of data size and of temporal window size on performance. Our experiments showed that OGC performs best, followed by OG and VE. RG was the slowest representation in all cases. We also observed that smaller temporal window sizes (and thus more windows to compute) lead to longer execution time for RG and VE.



(a) WikiTalk (b) SNB:300 (c) NGrams:M

Figure 16: aZoom^T - wZoom^T combination and switching between memory representations. Fixed data size, group-by cardinality and number of intervals, varying the size of windows, node quantifier ‘all’, edge quantifier ‘all’.

5.3 Operation Chaining

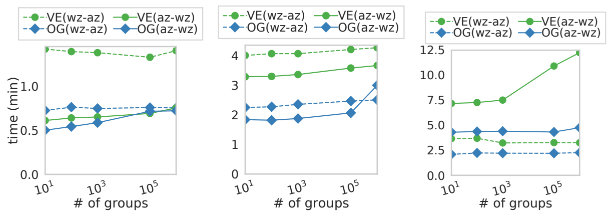
In this section we chain together aZoom^T to a wZoom^T and investigate whether switching between representations improves performance. Since OGC does not support attribute-based operation and due to the high memory usage and scalability issues of RG, we only run our experiments on VE and OG.

In the first experiment we run aZoom^T then wZoom^T with different windows sizes on WikiTalk, SNB:300 and NGrams:M. For aZoom^T on WikiTalk, we use edit count as the zoom attribute, for SNB we use first name, and for NGrams we use the word attribute. Figure 16 shows the results of this experiment. The x-axis lists window sizes for wZoom^T (in months for WikiTalk and SNB, and in years for NGrams), while the y-axis denotes the running time in minutes. Each line shows which representation is used. On WikiTalk, OG is the winner while OG-VE, VE-OG and VE are slightly slower. On SNB:300, VE-OG, and OG are fastest, and OG-VE is slowest, followed by VE.

In the previous section we saw that VE performs slightly better for aZoom^T on SNB, and OG performs significantly better for wZoom^T, so it makes sense for VE-OG and OG to show the best performance and for VE and OG-VE to show the worst. For NGrams, OG is the clear winner followed by OG-VE. The worst-performing combination here is VE-OG, followed by VE. On NGrams, OG performs significantly better for both aZoom^T and wZoom^T, and this can explain the results we are observing here.

In the next experiment we change the order of aZoom^T and wZoom^T. While this reordering does not always produce the same result, we can safely reorder the operations for WikiTalk and SNB, since no attributes change in these datasets, and so applying wZoom^T or aZoom^T first produces the same result with the “exist“ quantifier for both vertices and edges.

Figure 17 shows the effect of group-by cardinality on wZoom^T - aZoom^T and aZoom^T - wZoom^T. In this experiments, we load the full graph for each dataset, project each node attribute to a random value based on group-by cardinality, and then perform the operations, with window size set to 6 months for WikiTalk and SNB, and 10 years for NGrams. We vary group-by cardinality from 10 to 1 million. We observe an increase in the execution time as the group-by cardinality increases, which we attribute to the fact that aZoom^T produces a larger intermediate graph for cases where we perform aZoom^T first. In contrast, we see no significant change in the execution time when wZoom^T is executed first. Interestingly, performing wZoom^T first in NGrams yields faster running time. Unlike in WikiTalk and SNB, vertices in NGrams are not growth-only, and they also span over a longer



(a) WikiTalk (b) SNB:300 (c) NGrams:M

Figure 17: aZoom^T and wZoom^T performance for different group-by cardinalities with different zoom orders. Fixed data size and number of intervals. OG-based implementations perform best in most cases.

period of time. wZoom^T will reduce the number of snapshots and vertex tuples, which explains why wZoom^T - aZoom^T is faster than aZoom^T - wZoom^T.

Summary. We studied combining aZoom^T and wZoom^T for different combinations of parameters. Our experiments show that, while OG alone performs best in most cases, switching between representations does not significantly affect the running time. We also found that running aZoom^T before wZoom^T is fastest for growth-only datasets.

5.4 Summary

In this section, we first studied the effects of data size, the number of snapshots, the frequency of attribute change, and the group-by cardinality on the running time of aZoom^T. We observed that OG is the best performing representation for aZoom^T, followed by VE. We showed that representing the TGraph as a sequence of independent snapshots in RG results in the by far worst performance. The second part of this section focused on wZoom^T. We varied graph size and window size, and observed that OGC is the best-performing representation, followed by OG and VE. RG again exhibited the worst performance for wZoom^T. The last part of this section focused on combining aZoom^T and wZoom^T.

Overall, we found that OG, which balances temporal and structural locality, outperforms other representations in most cases.

6 RELATED WORK

Temporal models and languages in the relational literature are very mature (see, e.g., [10, 16, 23]). However, the same cannot be said for evolving graphs, where models differ in what time representation they adopt (point or interval), what top-level entities they model (graphs or sets of nodes and edges), whether they represent topology only or attributes or weights as well, and what types of evolution they support. Harary and Gupta [20] were, to the best of our knowledge, the first to informally propose to model graph evolution as a sequence of static graphs. This model has been predominant in the literature [15, 24–26, 38, 40], with various restrictions on the kinds of changes that can take place during graph evolution. In contrast to existing work, TGraph assigns periods of validity to nodes, edges and their properties, capturing evolution of graph topology and of node and edge attributes, and supports point-based semantics [37].

The attribute-based zoom operator is a temporal generalization of the node creation operator that is present in several conventional (non-temporal) graph query languages [42]. For example, StruQL outputs new nodes in a create clause, corresponding to the node creation operation with a Skolem function to create the object ids [13], while GOOD provides an abstraction operator that allows to create new nodes to represent multiple nodes based on shared properties [18]. To the best of our knowledge, a temporal generalization of this operator has not been considered except in our own prior work [37], and also has not been implemented in systems.

That said, the G* system supports SQL-style aggregation using the AggregateOperator per graph snapshot, supporting a limited version of summarization [26]. G* ingests evolving graph data one snapshot at a time, replicated across all machine without any compression, and is most similar to our RG. Our experiments showed that G* is not capable of loading graphs with a large number of intervals and does not scale for large size graphs [36]. We were not able to fully ingest any of our datasets used in Section 5 into G*. Chronograph, a system designed for temporal graph traversal [6], implements a version of temporal aggregation for the purpose of converting point-based to period-based semantics for edges, but not for nodes.

Temporal aggregation operators over relational data can be found in the literature, typically as an extension of non-temporal relational aggregation (see, e.g., example 10 in [11]). Li et al. proposed a general window aggregate for data streams [29] that can be applied to temporal relational data. Window aggregate semantics is based on a sliding window specification — a range and a slide — based on the desired data attribute that has a domain with a total order. The range specifies the width of the window e.g., 100 seconds or 100 rows, and the slide specifies how windows are formed. We are not aware of any proposal for an operator capable of changing the temporal resolution of evolving graphs, besides our own, introduced in [37], and no systems work on such an operator.

In our work we implement aZoom^T and wZoom^T operators in a dataflow system, and instantiate our ideas over Apache Spark [43], using the GraphX [17] library. We leverage the graph-specific optimizations provided by GraphX, as described in our implementation section, and incorporate temporal semantics into data representations and operators.

7 CONCLUSION

In this paper we proposed an implementation of two zoom operators — aZoom^T and wZoom^T — on evolving graphs. We detailed four physical representations — RG, VE, OG, and OGC, and described how to define the zoom operators using distributed dataflow operations, tailored to the corresponding data representation. We discussed how to efficiently implement the operators in Apache Spark with its GraphX library, and explained that our operator definitions could easily be implemented in other dataflow systems such as Apache Flink. In an extensive experimental evaluation on several real datasets with up to 1.3 billion edges, we explored the trade-offs in terms of temporal and structural locality with respect to zoom operator performance. We find that OG, which balances temporal and structural locality, outperforms the other representations in most cases.

In our future work we will extend our system to support additional operations on evolving graphs, such as Pregel-style analytics [30]. We will propose query optimization techinques for our workloads. Finally, we will design a query language with support for the proposed temporal zoom operators, among others.

REFERENCES

[1] Charu C. Aggarwal and Karthik Subbian. 2014. Evolutionary Network Analysis. *ACM Comput. Surv.* 47, 1 (2014), 10:1–10:36.
[2] Alexander Alexandrov et al. 2014. The stratosphere platform for big data analytics. *VLDB* 23, 6 (2014), 939–964.
[3] Renzo Angles et al. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.
[4] Michael H. Böhlen. 2009. Temporal Coalescing. In *Encyclopedia of Database Systems*. 2932–2936.
[5] Michael H Böhlen et al. 2009. Temporal Compatibility. In *Encyclopedia of Database Systems*. 2936–2939.
[6] Jaewook Byun et al. 2019. ChronoGraph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE TKDE* (2019).
[7] Paris Carbone et al. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.
[8] Jeffrey Chan et al. 2008. Discovering correlated spatio-temporal changes in evolving graphs. *Knowl. Inf. Syst.* 16, 1 (2008), 53–96.
[9] Junghoo Cho and H Garcia-Molina. 2000. The evolution of the web and implications for an incremental crawler. *VLDB* (2000), 200–209.
[10] Jan Chomicki. 1994. Temporal Query Languages: A Survey. In *ICTL*.
[11] Dignös et al. 2012. Temporal Alignment. In *SIGMOD*.
[12] Orri Erling et al. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD*. 619–630.
[13] Mary F. Fernández et al. 1997. A Query Language for a Web-Site Management System. *SIGMOD Record* 26, 3 (1997), 4–11.
[14] Mary F. Fernández et al. 2000. Declarative Specification of Web Sites with Strudel. *VLDB J.* 9, 1 (2000), 38–55.
[15] Afonso Ferreira. 2004. Building a reference combinatorial model for MANETs. *IEEE Network* 18, 5 (2004), 24–29. https://doi.org/10.1109/MNET.2004.1337732
[16] Shashi K Gadia and Chuen-Sing Yeung. 1988. A generalized model for a relational temporal database. In *ACM SIGMOD Record*, Vol. 17. ACM, 251–259.
[17] Joseph E. Gonzalez et al. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX*. 599–613.
[18] Marc Gyssens et al. 1994. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artif. Intell.* 66, 1 (1994), 57–89.
[19] Wentao Han et al. 2014. Chronos : A Graph Engine for Temporal Graph Analysis. In *EuroSys*.
[20] F. Harary and G. Gupta. 1997. Dynamic graph models. *Mathematical and Computer Modelling* 25, 7 (1997).
[21] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the SIGMOD*. 405–418.
[22] Ping-yu Hsu and D Stott Parker. 1995. Improving SQL with Generalized Quantifiers. In *ICDE*.
[23] Christian S. Jensen and Richard T. Snodgrass. 2009. Temporal Data Models. In *Encyclopedia of Database Systems*. 2952–2957.
[24] Andrey Kan et al. 2009. A Query Based Approach for Mining Evolving Graphs. In *AusDM 2009*, Vol. 101.
[25] Udayan Khurana and Amol Deshpande. 2016. Storing and Analyzing Historical Graph Data at Scale. In *EDBT*.
[26] Alan G. Labouseur et al. 2014. The G* graph database: efficiently managing large distributed dynamic graphs. *Distrib. and Parallel Databases* 33, 4 (2014).
[27] M. Lahiri and Berger-Wolf. 2008. Mining Periodic Behavior in Dynamic Social Networks. In *2008 Eighth IEEE ICDM*. 373–382.
[28] Timothy LaRock et al. 2019. Detecting Path Anomalies in Time Series Data on Networks. *arXiv preprint arXiv:1905.10580* (2019).
[29] Jin Li et al. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*.
[30] Grzegorz Malewicz et al. 2010. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*. 135–146.
[31] F Manola et al. 2013. RDF primer. W3C Recommendation 10, 1–107 (2004).
[32] Mauro San Martín et al. 2011. SNQL: A Social Networks Query and Transformation Language. In *AMW*.
[33] Frank McSherry et al. 2013. Differential Dataflow. In *CIDR 2013*,.
[34] Sergey Melnik et al. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. In *VLDB*.
[35] Youshan Miao et al. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Transactions on Storage* 11, 3 (2015), 14–34.
[36] Vera Z. Moffitt. 2017. *Framework for Querying and Analysis of Evolving Graphs*. Ph.D. Dissertation. Drexel University.
[37] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal graph algebra. In *Proceedings of DBPL 2017*. 10:1–10:12.
[38] Chenghui Ren et al. 2011. On Querying Historical Evolving Graph Sequences. *Proceedings of the VLDB Endowment* 4, 11 (2011), 726–737.
[39] Ingo Scholtes et al. 2016. Higher-order aggregate networks in the analysis of temporal networks: path structures and centralities. *The European Physical Journal B* 89, 3 (2016), 61.
[40] Konstantinos Semertzidis et al. 2015. TimeReach: Historical Reachability Queries on Evolving Graphs. In *EDBT*.
[41] Jun Sun and Jérôme Kunegis. 2016. Wiki-talk Datasets.
[42] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.
[43] Matei Zaharia et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.