Portal: A Query Language for Evolving Graphs

Vera Zaychik Moffitt
Drexel University
zaychik@drexel.edu

Julia Stoyanovich
Drexel University
stoyanovich@drexel.edu

ABSTRACT

Graphs are used to represent a plethora of phenomena, from the Web and social networks, to biological pathways, to semantic knowledge bases. Arguably the most interesting and important questions one can ask about graphs have to do with their evolution. Which Web pages are showing an increasing popularity trend? How does influence propagate in social networks? How does knowledge evolve?

In this paper we address the need to enable *systematic support* for scalable querying and analytics over *evolving graphs*. We propose a representation of an evolving graph, called a TGraph, which captures evolution of graph topology, and of attributes of vertices and edges, continuously through time. We develop a compositional TGraph algebra that includes such operations as temporal selection, subgraph, aggregation, and a rich class of analytics. We present Portal, a system that implements our model and algebra in scope of Apache Spark, an open-source distributed data processing framework. We develop multiple physical representations of evolving graphs and study the trade-offs between structural and temporal locality. We provide an extensive experimental evaluation on real datasets, demonstrating that careful engineering can lead to good performance.

1. INTRODUCTION

The importance of networks in scientific and commercial domains cannot be overstated. Considerable research and engineering effort is being devoted to developing effective and efficient graph representations and analytics. Efficient graph abstractions and analytics for *static graphs* are to researchers and practitioners in scope of open source platforms such as Apache Giraph, Apache Spark (through GraphX [14] and GraphLab (through the PowerGraph library [13]).

Analysis of *evolving graphs* has been receiving increasing attention [1, 8, 17, 23, 24, 26]. Yet, despite the recent activity, and despite increased variety and availability of evolving graph data, *systematic support for scalable querying and analytics over evolving graphs still lacks.* This support is urgently needed, due first and foremost to the scalability and efficiency challenges inherent in evolving graph analysis, but also to considerations of usability and ease of dissemination. *In this paper, we present Portal, a system for scalable exploratory analysis of evolving graphs, that fills this gap.*

Portal represents the evolution of a graph, including changes in topology or attribute values of vertices and edges, continuously through time using the TGraph abstraction. An example of a TGraph is given in Figures 1 and 2. Figure 1 gives the *representative graphs* of a TGraph— a sequence of graphs associated with a sequence of coalesced time periods. The semantics is that no change occurred in a graph for the duration of the time period. Figure 2 gives the *vertex-edge* representation of an evolving graph — a collection of coalesced temporal SQL relations with appropriate integrity constraints. The Portal system implements the TGraph abstraction and the operations of a fully compositional algebra, supporting exploratory analysis. Consider these questions over evolving graphs:

Which network nodes are showing an increasing (decreasing) popularity trend, or have increasing influence? Node popularity can be quantified by, e.g., node degree, centrality measures, or PageRank score. This information can be used to prioritize crawling on the Web, to target ads in a social network, and to captures the dynamics of zeitgeist in semantic knowledge bases. *Have any changes in network connectivity been observed, either suddenly or gradually?* Connectivity can be quantified as, e.g., pair-wise distance, length of shortest path between communities, or graph density. For networks describing insulin-based metabolism pathways, gradual pathway disruption can be used to determine the onset of type-2 diabetes [3]. For a website accessibility network, sudden loss of connectivity can signal that censorship is taking place. Portal supports efficient computation of node popularity and network connectivity measures with a combination of *temporal aggregation* and *temporal analytics.*

At what time scale can interesting trends be observed? The answer to this question may not be known a priori, at the time when graph evolution data is being recorded. Changes in node centrality in a social network may be observable on the scale of weeks, but not months. On the Web, periodic events may change popularity of websites, with observable trends on the scale of days, but not hours or months. Furthermore, the same network may exhibit different trends at different time scales, e.g., node popularity may change at a different rate than over-all network density. Portal supports this analysis via *temporal aggregation.*

Proceedings of the VLDB Endowment, Vol. 10, No. 3
Copyright 2016 VLDB Endowment 2150-8097/16/11.

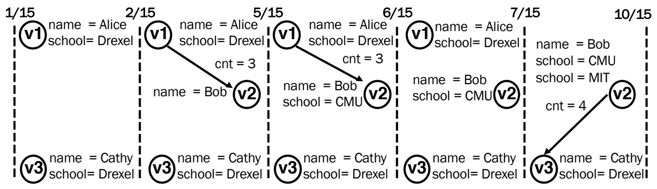


Figure 1: Representative graphs of TGraph T1.

Can information about graph evolution be used to make graph analytics more stable? Algorithms that compute website popularly can be vulnerable to link spam, but the identity of spammers is transient [27]. Persistence vs. transience of a node, edge, or, more generally, of a subgraph, is a meaningful aspect of quality. The *temporal aggregation* operation in Portal can be used to find representative subgraphs of an evolving graph.

How can multiple data sources be used jointly to complement or corroborate information about graph evolution? Multiple datasets may be available, each describing a series of crawls of different but possibly overlapping portions of the Web graph. Further, network states may be recorded at different, possibly overlapping, time periods, or at different temporal scales. Can these datasets be unified to support analysis of network evolution trends? Portal supports *union* and *intersection* operations over evolving graphs that enable this analysis.

Contributions. We make the following contributions to support efficient and usable analysis of evolving graphs.

1. We propose a novel representation of an evolving graph, called a TGraph, which captures the evolution of both graph topology and vertex and edge attributes (Section 2).
2. We define a fully compositional TGraph algebra, which includes temporal selection, subgraph, map, aggregation, join, union, and a rich class of analytics (Section 3).
3. We develop several physical representations of the logical TGraph data structure corresponding to different tradeoffs in temporal and structural locality, and implement these representations and the operations of TGraph algebra in Apache Spark, leveraging the GraphX framework [14] (Section 4).
4. We conduct an extensive experimental evaluation with real datasets, demonstrating that Portal scales (Section 5).

2. MODEL

Following the SQL:2011 standard [19], we adopt the *closed-open* period model, where a period (or interval) represents a continuous set of time instances, starting from and including the start time, continuing to but excluding the end time. Time instances, or chronons, have limited precision and the time domain has total order.

Definition 2.1 (Time period). *A* time period $p = [start, end)$ *is an interval of the continuous time domain, subject to the constraint* $start < end$.

We quantify relationships between time periods using the following Allen's relations [2] with equality: meets, overlaps, starts, finishes, during, and equals. We will use p contains q as a shorthand for p starts $q \vee p$ finishes $q \vee p$ during $q \vee p$ equal q. We denote by $p \cap q$ the intersection of p and q.

2.1 Representative graphs of a TGraph

The basic building block of our model is a TGraph, which represents a single graph that evolves continuously over time,

TV (v, p)

v	p
v1	[1/15, 7/15)
v2	[2/15, 10/15)
v3	[1/15, 10/15)

TA^V (v, p, a)

v	p	a
v1	[1/15, 7/15)	name=Alice school=Drexel
v2	[2/15, 5/15)	name=Bob
v2	[5/15, 7/15)	name=Bob school=CMU
v2	[7/15, 10/15)	name=Bob school=CMU school=MIT
v3	[1/15, 10/15)	name=Cathy school=Drexel

TE (v1, v2, p)

v1	v2	p
v1	v2	[2/15, 6/15)
v2	v3	[7/15, 10/15)

TA^E (v1, v2, p, a))

v1	v2	p	a
v1	v2	[2/15, 6/15)	cnt=3
v2	v3	[7/15, 10/15)	cnt=4

Figure 2: Vertex-edge representation of TGraph T1.

and is defined over a set of vertices V, edges $E \subseteq V \times V$, vertex attributes A^V, and edge attributes A^E.

Definition 2.2 (TGraph). *A TGraph* $\mathsf{T}^{\mathsf{RG}}(g, p)$ *is a valid-time period-based relation that associates a state of the graph* g *with a time period* p *during which the graph is in that state. Each* $g = (V_g, E_g, \mathsf{A}^\mathsf{V}{}_g, \mathsf{A}^\mathsf{E}{}_g)$*, with* $V_g \subseteq V$, $E_g \subseteq E$, $\mathsf{A}^\mathsf{V}{}_g \subseteq \mathsf{A}^\mathsf{V}$, $\mathsf{A}^\mathsf{E}{}_g \subseteq \mathsf{A}^\mathsf{E}$. T^{RG} *is temporally coalesced:*

$$\forall \mathsf{T}^{\mathsf{RG}}(g, p) \nexists \mathsf{T}^{\mathsf{RG}}(g, p') \mid \quad p \text{ meets } p' \vee p \text{ contains } p' \vee p \text{ overlaps } p' \quad (1)$$

T^{RG} *represents evolution of a single graph over time:*

$$\forall \mathsf{T}^{\mathsf{RG}}(\_, p) \nexists \mathsf{T}^{\mathsf{RG}}(\_, p') | \; p \text{ contains } p' \vee p \text{ overlaps } p' \quad (2)$$

We refer to each g *that appears in some tuple* (g, p) *of* T^{RG} *as a representative graph of* T^{RG} *during period* p.

An example of a TGraph is given in Figure 1, where 5 representative graphs (states) of T1 are associated with 5 consecutive periods $p_1 = [1/15, 2/15), \ldots, p_5 = [7/15, 10/15)$.

Vertices and edges of a TGraph are not required to be homogeneous in terms of their schemas. In line with several popular graph databases, we use the property graph model [25] to represent vertex and edge attributes. Each vertex and edge of g during period p is associated with a (possibly empty) *bag* of properties, and each property is represented by a key-value pair. Values are not restricted to be of atomic types, and may, e.g., be maps or tuples.

As is typical in interval-based models, each representative graph is assumed to exists continuously, and remains unchanged, during the associated period. By this we mean that graph topology (identities of its vertices and edges), and properties of each vertex and edge remain unchanged. The statement that T^{RG} is temporally coalesced means that each fact (unchanging state of the graph) is represented exactly once for each time period of maximal length when it holds [4]. For interval-based models, requiring that relations be coalesced is both space-efficient and avoids semantic ambiguity (see [16] Fig. 2 and its description).

2.2 Vertex-edge representation of a TGraph

We now give an alternative representation of a TGraph that uses valid-time temporal SQL relations [4], with nesting, to represent evolution of graph topology and of its vertex and edge attributes. We term this logical representation the *vertex-edge TGraph*. An example is given in Figure 2. We will refer to the representative graphs (Definition 2.2) and vertex-edge (Definition 2.3) representations of a TGraph jointly as T, and will disambiguate between the two representations with T^{RG} and T^{VE} when necessary.

Definition 2.3 (Vertex-Edge TGraph). *A vertex-edge representation of TGraph is a pair* $\mathsf{T}^{\mathsf{VE}} = (TV, TE)$. *TV is a valid-time temporal SQL relation with schema* $TV(\underline{v}, \underline{p})$ *that associates a vertex with the time period during which it is present. TE is a valid-time temporal SQL relation with schema* $TE(\underline{v_1}, \underline{v_2}, \underline{p})$*, connecting pairs of vertices from TV. TV and TE are temporally coalesced:*

$$\forall TV(v,p) \ \ \nexists TV(v,p') \ \ | \qquad p \ \mathsf{meets}\ p' \ \lor \ p \ \mathsf{contains}\ p' \ \lor \ p \ \mathsf{overlaps}\ p' \quad (3)$$

$$\forall TE(v_1,v_2,p) \ \ \nexists TE(v_1,v_2,p') \ \ | \qquad p \ \mathsf{meets}\ p' \ \lor \ p \ \mathsf{contains}\ p' \ \lor \ p \ \mathsf{overlaps}\ p' \quad (4)$$

An edge connects a pair of vertices that exist at the time when the edge exists:

$$\forall TE(v_1,v_2,p) \ \ \exists TV(v_1,p_1), TV(v_2,p_2) \ \ | \qquad p_1 \ \mathsf{contains}\ p \ \land \ p_2 \ \mathsf{contains}\ p \quad (5)$$

T^{VE} *optionally includes vertex and edge attribute relations* TA^{V} *and* TA^{E}. *Both are coalesced, and associate bags of properties with existing vertices and edges.*

$$\forall \mathsf{TA}^{\mathsf{V}}(v,p,a) \ \ \nexists \mathsf{TA}^{\mathsf{V}}(v,p',a) \ \ | \qquad p \ \mathsf{meets}\ p' \ \lor \ p \ \mathsf{contains}\ p' \ \lor \ p \ \mathsf{overlaps}\ p' \quad (6)$$

$$\forall \mathsf{TA}^{\mathsf{V}}(v,p,a) \qquad \exists TV(v,p') \qquad | \qquad p' \ \ \mathsf{contains} \ \ p \quad (7)$$

$$\forall \mathsf{TA}^{\mathsf{E}}(v_1,v_2,p,a) \ \ \nexists \mathsf{TA}^{\mathsf{E}}(v_1,v_2,p',a) \ \ | \qquad p \ \mathsf{meets}\ p' \ \lor \ p \ \mathsf{contains}\ p' \ \lor \ p \ \mathsf{overlaps}\ p' \quad (8)$$

$$\forall \mathsf{TA}^{\mathsf{E}}(v_1,v_2,p,a) \ \ \exists TE(v_1,v_2,p') \ \ | \ \ p' \ \mathsf{contains}\ p \quad (9)$$

Graphs may be directed or undirected. For undirected graphs we choose a canonical representation of an edge, with $v_1 \leq v_2$ (self-loops are allowed).

Conditions 3 and 4 in Definition 2.3 state that TV and TE are coalesced [4], i.e., that each vertex and edge is represented exactly once for each time period of maximal length when it is present. Conditions 6 and 8 state that all attribute relations are coalesced, i.e., that an attribute is represented exactly once for each time period of maximal length in which its value did not change. Consider Figure 2 and note that there is a single tuple for v_2 in TV, but three tuples for v_2 in TA^{V}, because $school = CMU$ was added at time 5/15, and $school = MIT$ at 7/15. It is not required that a vertex or an edge be represented in TA^{V} and TA^{E}— a vertex or edge that never had any associated properties will have no corresponding tuples in TA^{V} or TA^{E}.

Definitions 2.2 and 2.3 give two alternative views of graph evolution. In Definition 2.2 the graph is represented in its entirety, with a change of state occurring whenever there is a change in graph topology or in its vertex or edge properties. In other words, time periods here are at their finest granularity. In contrast, Definition 2.3 decouples evolution of graph vertices, edges and attribute values, coalescing each relation independently of the others. This makes graph maintenance and manipulation more manageable.

The relations of T^{VE} can be implemented in temporal SQL, as per the SQL:2011 standard [19], but this implementation will not ensure that relations are coalesced. SQL:2011 does not provide a way to specify that a relation be coalesced, and does not enable efficient mechanisms to implement the coalesce operator. Coalescing requires that the system automatically merge adjacent and overlapping time periods. This operation, which is similar to duplicate elimination in conventional databases, has been extensively studied in the literature [4, 29], but is not supported by the standard.

While our vertex-edge representation is based on temporal SQL, we reiterate that non-key attributes of vertices and edges are stored as collections of properties. Definition 2.3 presents a logical data structure and may be implemented, e.g., by a columnar representation of vertex and edge properties (each property in a separate relation), or by some hybrid representation. A columnar representation may be more efficient if different properties change at different rates.

Our choice to use attribute relations is in contrast to representing vertex and edge attributes as part of TV and TE. The main reason is to streamline the enforcement of referential integrity constraints of Definition 2.3. Consider again the example in Figure 2. If vertex attributes were stored as part of TV, then there would be two tuples for v_2 in this relation whose validity periods overlap with that of edge $e(v_1, v_2)$ — one for each $[2/15, 5/15)$ and $[5/15, 7/15)$. This would in turn require that $e(v_1, v_2)$ be mapped to two tuples in TV as part of referential integrity checking on v_2. Matching a tuple with a set of tuples in the referenced table, while supported by the SQL:2011 standard, is potentially inefficient, and we avoid it in our representation. Another reason for optional TA^{V} and TA^{E} is that in many cases we are interested in applying operations (e.g., analytics) only to graph topology, and in that case TV and TE are sufficient.

2.3 Switching between representations

T^{RG} and T^{VE} are equivalent in terms of the information they contain. T^{RG} can be computed from T^{VE} by (1) enumerating all time points during which some change occurred, i.e., those that appear as either the start or the end of some time period in TV, TE, TA^{V}, and TA^{E}; (2) computing T^{RG} periods by generating pairs of adjacent time points from step 1, and sorting them by *start*; (3) constructing a graph g for each time period p by selecting tuples from TV, TE, TA^{V}, and TA^{E} that overlap with p.

To compute T^{VE} from T^{RG}, we iterate over $\mathsf{T}^{\mathsf{RG}}(g, p)$, computing $\mathsf{TV}_g(v, p)$ (similarly for TE_g, ${\mathsf{TA}^{\mathsf{V}}}_g$, ${\mathsf{TA}^{\mathsf{V}}}_g$) for each tuple, collecting $\mathsf{TV}_g \rightarrow \mathsf{TV}$, $\mathsf{TE}_g \rightarrow \mathsf{TE}$, ${\mathsf{TA}^{\mathsf{V}}}_g \rightarrow \mathsf{TA}^{\mathsf{V}}$, ${\mathsf{TA}^{\mathsf{E}}}_g \rightarrow \mathsf{TA}^{\mathsf{E}}$, and coalescing each TV, TE, TA^{V}, TA^{E}.

3. ALGEBRA

In this section we describe the operators of TGraph algebra. The algebra is compositional: all operators take a TGraph or a pair of TGraphs as input, and produce a TGraph. Our algebra is based on the principle of snapshot reducibility [5], which states that an operation should yield the same result when applied to the temporal representation as it would if it were applied separately to every snapshot state of the temporal representation.

Böhlen et al. [4] show that temporal selection, Cartesian product and difference all produce a coalesced relation as output if the input was coalesced. They also show that temporal union and temporal projection can give rise to an uncoalesced output even if the inputs were coalesced. Intuitively, this is because union and projection can give rise

Algorithm 1 Evaluation of a unary operation op on T^{VE}

```
Require: TGraph $\mathsf{T}^{\mathsf{VE}}(\mathsf{TV};\mathsf{TE};\mathsf{TA}^{\mathsf{V}};\mathsf{TA}^{\mathsf{E}})$, operation op.
1: $\mathsf{TV}' = \mathsf{coal}(\mathsf{op}(\mathsf{TV}))$
2: $\mathsf{TE}' = \mathsf{coal}(\mathsf{op}(\mathsf{TE}))$
3: $\mathsf{TA}^{\mathsf{V}'} = \mathsf{coal}(\mathsf{op}(\mathsf{TA}^{\mathsf{V}}))$
4: $\mathsf{TA}^{\mathsf{E}'} = \mathsf{coal}(\mathsf{op}(\mathsf{TA}^{\mathsf{E}}))$
5: enforce foreign keys on $\mathsf{TE}'$ w.r.t. $\mathsf{TV}'$
6: enforce foreign keys on $\mathsf{TA}^{\mathsf{V}'}$ w.r.t. $\mathsf{TV}'$
7: enforce foreign keys on $\mathsf{TA}^{\mathsf{E}'}$ w.r.t. $\mathsf{TE}'$
8: return new $\mathsf{T}^{\mathsf{VE}}(\mathsf{TV}';\mathsf{TE}';\mathsf{TA}^{\mathsf{V}'};\mathsf{TA}^{\mathsf{E}'})$
```

to duplicates in traditional relational algebra, and lack of coalescing is the temporal analogy to a duplicate. These observations also hold in our scenario at the level of individual relations, each containing representative graphs, vertices, edges, or vertex/edge attributes.

Our algebra operates on TGraphs, and so, in addition to keeping the data structure (and its constituent parts) coalesced, we must ensure that the result is a valid TGraph. Importantly, when manipulating the vertex-edge representation, we do not require that each intermediate state of the data structure correspond to a valid TGraph, but rather that the final result, which is usually derived after several steps, be valid. To have a useful algebra, we do not reject a change that would lead to a violation in referential integrity. Instead, we compute a consistent result by removing the tuples that violate referential integrity.

Algorithm 1 outlines the evaluation of a unary operation op (T^{VE}), overloading op as appropriate when applying the operation to constituent parts of T^{VE}. Some of the steps in Algorithm 1 may be unnecessary because of the properties of the particular operation, as we will see in the remainder of this section. Furthermore, some operations may produce correct results (up to coalescing) *even when computing over uncoalesced inputs.* We will revisit this point in Section 4.1.

3.1 Slice

The unary *slice* operator, denoted $\tau_c(\mathsf{T})$, where c is a time period, cuts a temporal slice from T. The resulting TGraph will contain representative graphs whose period p has a non-empty intersection with c. If $p.start < c.start$ or $p.end > c.end$ for some tuple (g,p), then p is trimmed to be within the boundaries of c: $\tau_c(\mathsf{T}^{\mathsf{RG}}) = \{(g, p \cap c) \mid (g,p) \in \mathsf{T}^{\mathsf{RG}} \wedge (c \text{ overlaps } p \vee c \text{ contains } p)\}$. To evaluate $\tau_c(\mathsf{T}^{\mathsf{VE}})$, we apply τ_c to each of the four constituent relations of T^{VE}: $\tau_c(\mathsf{TV}) = \{(v, p \cap c) \mid (v,p) \in \mathsf{TV} \wedge (c \text{ overlaps } p \vee c \text{ contains } p)\}$, and analogously for each TE, TA^{V} and TA^{E}.

Slice does not uncoalesce. Whether evaluated over T^{RG} or T^{VE}, slice is guaranteed to return a coalesced relation when evaluated over a coalesced input. This is because, for any relation R, there will be at most one tuple from R in the result of $\tau_c(R)$, with a validity period that is either the same as it was in R, or further restricted (trimmed). Therefore, there is no need to coalesce T^{RG} after slice, or on lines 1-4 of Algorithm 1 when operating over T^{VE}.

Slice does not require FK enforcement for T^{VE}. To see why, consider an edge $\mathsf{TE}(v_1, v_2, p)$ and one of the corresponding vertices $\mathsf{TV}(v_1, p_1)$, such that p_1 contains p (per Definition 2.3 condition 5). Suppose now that slice was applied to TV and to TE with condition c. Is it possible that edge $(v_1, v_2, p \cap c)$ is in the result of $\tau_c(\mathsf{TE})$ (i.e., $p \cap c \neq \emptyset$), while vertex $\mathsf{TV}(v_1, p_1 \cap c)$ is not in the result of $\tau_c(\mathsf{TV})$ (i.e., $p_1 \cap c = \emptyset$)? Clearly, the answer is no, since p_1 contains p, and so it must be the case that p_1 contains $(p \cap c)$. A similar argument justifies that FK enforcement is not needed for $\tau_c(\mathsf{TA}^{\mathsf{V}})$ (w.r.t. $\tau_c(\mathsf{TV})$) and for $\tau_c(\mathsf{TA}^{\mathsf{E}})$ (w.r.t. $\tau_c(\mathsf{TE})$).

3.2 Temporal subgraph matching

Temporal subgraph matching is defined analogously to subgraph matching in non-temporal graphs: it applies a subgraph function f to every representative graph of the input. To ensure that a valid TGraph is computed as a result of this operation, we restrict our attention to functions that compute a single subgraph of a given representative graph as a result: $\sigma_f(\mathsf{T}^{\mathsf{RG}}) = \{(g', p) \mid (g,p) \in \mathsf{T}^{\mathsf{RG}} \wedge g' = f(g) \wedge V_{g'} \subseteq V_g \wedge E_{g'} \subseteq E_g\}$. Even more specifically, we focus on functions that can be expressed as a pair of *conjunctive queries* $\sigma_{C_V, C_E}(\mathsf{T}^{\mathsf{RG}})$, where C_V specifies *non-temporal predicates* over the vertices, and C_E — over the edges. (Computing arbitrary subgraphs of an evolving graph is beyond the scope of this paper, and we defer this to future work.)

Like other unary operators, $\sigma_{C_V, C_E}(\mathsf{T}^{\mathsf{VE}})$ follows the outline of Algorithm 1. Since C_V and C_E may involve predicates over the attributes, we compute the join of the vertex (resp. edge) relation with the corresponding attribute relation and push selections:$\mathsf{TV}' = \pi_{v,p}(\sigma_{C_{V1}}(V) \bowtie \sigma_{C_{V2}}(\mathsf{TA}^{\mathsf{V}}))$ (line 1),$\mathsf{TE}' = \pi_{v_1,v_2,p}(\sigma_{C_{E1}}(E) \bowtie \sigma_{C_{E2}}(\mathsf{TA}^{\mathsf{E}}))$ (line 2), $\mathsf{TA}^{\mathsf{V}'} = \sigma_{C_{V2}}(\mathsf{TA}^{\mathsf{V}})$ (line 3), $\mathsf{TA}^{\mathsf{E}'} = \sigma_{C_{V2}}(\mathsf{TA}^{\mathsf{E}})$ (line 4).

Subgraph may uncoalesce T^{RG}. Consider T1 in Figure 1. The query $\sigma_{C_V:school='Drexel', C_E:\top}(\mathsf{T1})$ matches vertices v_1 and v_3 in every representative graph in which these occur. Since graphs for time periods $[1/15, 2/15)$ through $[6/15, 7/15)$ are identical, the result will be uncoalesced, and will need to be coalesced explicitly. The final result will consist of 2 representative graphs, with both v_1 and v_3 for $[1/15, 7/15)$, and with v_3 only for $[7/15, 10/15)$.

Subgraph does not uncoalesce T^{VE}. Consider again the computation of TV' described above, with a query that involves projection, selection and join over temporal SQL relations TV and TA^{V}. While selection and join cannot produce an uncoalesced output if the input is coalesced, projection may produce an uncoalesced output relation [4]. Interestingly, projection does not result in an uncoalesced output in this case. To see why, suppose that C_V is trivial, i.e., that $\sigma_{C_{V1}}(\mathsf{TV}) = \mathsf{TV}$ and $\sigma_{Q_{C2}}(\mathsf{TA}^{\mathsf{V}}) = \mathsf{TA}^{\mathsf{V}}$. Then $\mathsf{TV}' = \pi_{v,p}(\mathsf{TV} \bowtie \mathsf{TA}^{\mathsf{V}})$, and since $\mathsf{TV} \bowtie \mathsf{TA}^{\mathsf{V}}$ is a primary key-foreign key join, then $\mathsf{TV}' = \mathsf{TV}$. If C_V is non-trivial, i.e., $\sigma_{Q_{C1}}(\mathsf{TV}) \subset \mathsf{TV}$ or $\sigma_{C_{V2}}(\mathsf{TA}^{\mathsf{V}}) \subset \mathsf{TA}^{\mathsf{V}}$, then it will be the case that $V' \subset V$. In both cases, if TV is coalesced then so is TV'. A similar argument applies to the edges relation TE'. Finally, since $\mathsf{TA}^{\mathsf{V}'}$ and $\mathsf{TA}^{\mathsf{E}'}$ are computed from coalesced input relations using only selection, they are guaranteed to be coalesced. Thus, it is not necessary to coalesce on lines 1-4 of Algorithm 1.

Subgraph requires FK enforcement for T^{VE}. A natural query specifies a selection condition over the vertices, and computes the vertex-induced subgraph. In this case we cannot compute TE' from TE alone, but will need to remove edges for which one or both vertices are not present in TV'. Similarly, we must remove tuples from $\mathsf{TA}^{\mathsf{V}'}$ and $\mathsf{TA}^{\mathsf{E}'}$ for which no corresponding tuples exist in TV' and TE'.

3.3 Temporal map

Temporal map iterates over the tuples of T^{RG}, and applies the user-specified map functions M_V and M_E to the vertices and edges of each g: $\mathsf{map}_{M_V,M_E}(\mathsf{T^{RG}}) = \{(g', p) \mid (g, p) \in \mathsf{T^{RG}} \wedge g' = \mathsf{map}_{M_V,M_E}(g)\}$. To evaluate $\mathsf{map}_{M_V,M_E}(\mathsf{T^{VE}})$, we set $\mathsf{TV}' = \mathsf{TV}$ and $\mathsf{TE}' = \mathsf{TE}$, and compute $\mathsf{TA^{V}}' = \mathsf{map}_{M_V}(\mathsf{TA^V})$ and $\mathsf{TA^{E}}' = \mathsf{map}_{M_E}(\mathsf{TA^E})$.

While map is an arbitrary user-specified function, there are some common use cases. Map can be used to remove vertex and edge properties, as in projection. In such cases we will use short-hand notation similar to projection, listing the properties that we wish to retain. For example, $\mathsf{map}_{M_V:school,M_E:\emptyset}(\mathsf{T1})$ will keep only the school property of the vertices, and no properties of the edges. Another useful map operation eliminates duplicates in the bag of vertex or edge properties. It may also be useful to flatten nested bags or aggregate multiple values of the same property of a vertex or edge, e.g., compute a sum or an average following temporal intersection or union (Section 3.5).

Map may uncoalesce $\mathsf{TA^V}$, $\mathsf{TA^E}$ and $\mathsf{T^{RG}}$. Consider computing $\mathsf{map}_{m_V:name,M_E:\emptyset}(\mathsf{T1})$ over $\mathsf{T1}$ in Figure 2. There will be three identical tuples in the result for vertex v_2 for $[2/15, 5/15)$, $[5/15, 7/15)$ and $[7/15, 10/15)$, which must be coalesced to return a valid $\mathsf{TA^V}$. A similar argument holds for $\mathsf{TA^E}$. This operation will also produce two identical representative graphs in $\mathsf{T^{RG}}$ in Figure 1 for $[2/15, 5/15)$ and $[5/15, 6/15)$, which will have to be coalesced explicitly.

Map does not require FK enforcement for $\mathsf{T^{VE}}$. This is because only $\mathsf{TA^V}$ and $\mathsf{TA^E}$ are affected, while the contents of TV and TE remain as in the input.

3.4 Temporal aggregation

We argued in the introduction that it is interesting and insightful to analyze an evolving graph at different levels of granularity. For example, the user may want to aggregate multiple consecutive representative graphs into a single representative graph, coarsening the granularity, or to predefine temporal resolution and look at the graph at that scale, irrespective of whether this resolution happens to be finer or coarse than the natural evolution rate of the graph. For this, we will use *moving window temporal aggregation*. Our approach is inspired by stream aggregation work of [21], adopted to graphs, and by generalized quantifiers of [15].

Temporal aggregation is denoted $\gamma_{W,Q_V,Q_E,A_V,A_E}(\mathsf{T})$, where W is the window specification, Q_V and Q_E are vertex and edge aggregation quantifiers, and A_V and A_E are the optional aggregation functions. It produces a consolidated evolving graph with specific temporal granularity.

Window specification W is of the form n $\{unit|\mathsf{changes}\}$, where n is an integer, and *unit* is a time unit, e.g., 10 *min*, 3 *years*, or any multiple of the usual time units. Window specification of the form n changes defines the window in terms of change over $\mathsf{T^{RG}}$. For example, $W = 3$ changes will aggregate sequences of 3 representative graphs into 1. Window boundaries are computed left-to-right, i.e., from least to most recent. The right-most window may correspond to fewer than n representative graphs from the input. Our window specification by change is similar to slide-by-row window in stream aggregation [21]. Note that, because TGraph algebra is compositional, we do not support temporal aggregation with overlapping windows, because it does not produce a valid TGraph. Also unlike [21], we do not currently support aggregation simultaneously by time and by non-temporal attributes (e.g., vertex properties). Incorporating this into TGraph algebra is in our immediate plans.

Algorithm 2 Temporal aggregation in $\mathsf{T^{VE}}$.

```
Require: $\mathsf{T^{VE}}$ ($\mathsf{TV}$;$\mathsf{TE}$;$\mathsf{TA^V}$;$\mathsf{TA^E}$), window specification $W$,
    vertex quantifier $Q_V$, edge quantifier $Q_E$, vertex aggre-
    gate function $A_V$, vertex aggregate function $A_E$.
1: $P = \mathsf{computePeriods}(W, \mathsf{TV}, \mathsf{TE}, \mathsf{TA^V}, \mathsf{TA^E})$
2: $\mathsf{TV}' = \mathsf{coal}(\gamma_{P,Q_V}(\mathsf{TV}))$
3: $\mathsf{TE}' = \mathsf{coal}(\gamma_{P,Q_E}(\mathsf{TE}))$
4: $\mathsf{TA^V}' = \mathsf{coal}(\gamma_{P,A_V}(\mathsf{TA^V}))$
5: $\mathsf{TA^E}' = \mathsf{coal}(\gamma_{P,A_E}(\mathsf{TA^E}))$
6: follow steps 5-7 of Algorithm 1
7: return new $\mathsf{T^{VE}}(\mathsf{TV}'; \mathsf{TE}'; \mathsf{TA^V}'; \mathsf{TA^E}')$
```

Aggregation quantifiers Q_V and Q_E are of the form { all | most | at least n | exists }, where n is a decimal representing the percentage of the time during which an entity (vertex or edge) existed, relative to the duration of the aggregation window. These are useful for producing different kinds of representative graphs. For example, to produce representative graphs with only strong connections over a volatile evolving graph, we may want to only include vertices that span the entire aggregation window ($Q_V = \mathsf{all}$), and edges that span a large portion of the window ($Q_E = \mathsf{most}$).

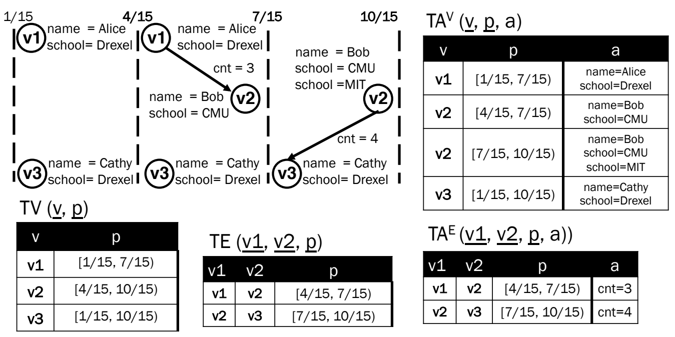
The optional *aggregation functions* A_V and A_E compute new values for vertex and edge properties representative of the whole window, e.g., $A_V = \mathsf{any}(name), \mathsf{last}(school)$ and $A_E = \mathsf{sum}(cnt)$. We support the standard { count | min | max | sum | average }, which have their customary meaning. We also support { any | first | last | trend | list }, which are possible to compute because properties being aggregated have temporal information. first and last refer to earliest/latest non-null value of the given property in the window, while trend computes the slope of the least squares line using linear regression, making an adjustment when a value is missing, and list associates with a key an ordered collection of values (but does not keep the associated validity periods).

Key-value pairs for vertex and edge properties for which no aggregation functions are specified, are collected into a bag corresponding to the entity in the result. These can be subsequently transformed with temporal map (Section 3.3).

Temporal aggregation over $\mathsf{T^{VE}}$ follows the outline of Algorithm 1, but requires an additional step, and is revisited in Algorithm 2. We compute aggregation periods based on window specification in line 1. several aggregation periods in lines 2—5. Next, as part of $\gamma_{P,Q_V}(\mathsf{TV})$ computation, we assign each tuple from TV to an aggregation period, group vertices by v and evaluate Q_V on each group. Only groups for which Q_V evaluates to true are retained. Edges are processed similarly by $\gamma_{P,Q_E}(\mathsf{TE})$ on line 3. Lines 4 and 5 compute $\gamma_{P,A_V}(\mathsf{TA^V})$ and $\gamma_{P,A_E}(\mathsf{TA^E})$, respectively, by computing a group for each vertex or edge within the period, and applying the aggregate functions over each group.

Temporal aggregation over $\mathsf{T^{RG}}$ is computed by first calculating time periods from W and $\mathsf{T^{RG}}$, and then processing the representative graphs directly.

Figure 3a illustrates temporal aggregation by time ($W = 3$ months), and Figure 3b — by change ($W = 3$ changes). Both are applied to T1 in our running example, and list the same aggregation quantifiers (all for vertices and exists for edges) and aggregate functions (first for vertex and edge properties). v_2 is present in the result in Figure 3a starting at 4/15 because it did not exist for the entirety of the first window, while in Figure 3b it is produced starting 6/15.


1/15
4/15
7/15
10/15
v1 name = Alice school= Drexel
v1 name = Alice school= Drexel
cnt = 3
name = Bob school = CMU v2
name = Bob school = CMU school =MIT v2
cnt = 4
v3 name = Cathy school= Drexel
v3 name = Cathy school= Drexel
v3 name = Cathy school= Drexel

TA^V (v, p, a)

v	p	a
v1	[1/15, 7/15)	name=Alice school=Drexel
v2	[4/15, 7/15)	name=Bob school=CMU
v2	[7/15, 10/15)	name=Bob school=CMU school=MIT
v3	[1/15, 10/15)	name=Cathy school=Drexel

TV (v, p)

v	p
v1	[1/15, 7/15)
v2	[4/15, 10/15)
v3	[1/15, 10/15)

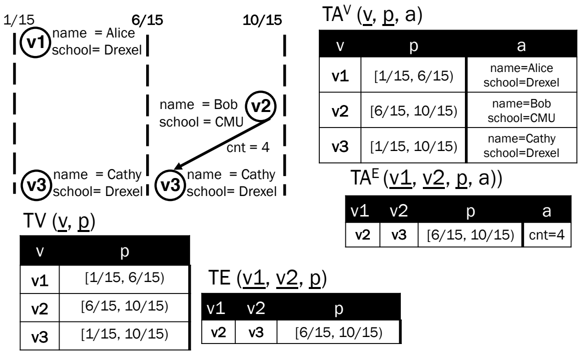
TE (v1, v2, p)

v1	v2	p
v1	v2	[4/15, 7/15)
v2	v3	[7/15, 10/15)

TA^E (v1, v2, p, a))

v1	v2	p	a
v1	v2	[4/15, 7/15)	cnt=3
v2	v3	[7/15, 10/15)	cnt=4

(a) Aggregation of T1 by time: $W = 3$ months.


1/15
6/15
10/15
v1 name = Alice school= Drexel
name = Bob school = CMU v2
cnt = 4
v3 name = Cathy school= Drexel
v3 name = Cathy school= Drexel

TA^V (v, p, a)

v	p	a
v1	[1/15, 6/15)	name=Alice school=Drexel
v2	[6/15, 10/15)	name=Bob school=CMU
v3	[1/15, 10/15)	name=Cathy school=Drexel

TA^E (v1, v2, p, a))

v1	v2	p	a
v2	v3	[6/15, 10/15)	cnt=4

TV (v, p)

v	p
v1	[1/15, 6/15)
v2	[6/15, 10/15)
v3	[1/15, 10/15)

TE (v1, v2, p)

v1	v2	p
v2	v3	[6/15, 10/15)

(b) Aggregation of T1 by change: $W = 3$ changes.

Figure 3: Aggregation, $Q_V =$ all, $Q_E =$ exists, $A_V =$ first, $A_E =$ first.

Temporal aggregation may uncoalesce T^{VE} and T^{RG}. Consider T that consists of only 1 vertex v_1 and no edges, and suppose that v_1 exists during the odd months of the year. TV contains $(v_1, [1/16, 2/16)), \ldots, (v_1, [11/16, 12/16))$, for which time periods do not meet and so these tuples cannot be coalesced. Next consider $\mathsf{T}^{\mathsf{VE}\prime}$ in the result of $\gamma_{W=2\ months, Q_V=exists, Q_E=all}(T)$. This relation will contain $(v_1, [1/16, 3/16)), \ldots,\ , (v_1, [11/16, 1/17))$ — one tuple per input tuple, all tuples meet and so TV' must be coalesced. A similar argument holds for, TE, TA^{V}, TA^{E}, T^{RG}.

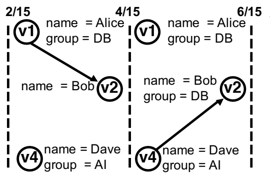
Temporal aggregation requires FK enforcement for T^{VE}. As illustrated in Figure 3a, $(v_1, v_2, [1/15, 4/15))$ is absent from TE' because, although this edge was present in TV for part of the period, and so meets the condition $Q_E =$ exists, vertex $(v_2, [1/15, 5/15))$ is absent from TV'.

We now give an analysis of whether FK enforcement is required, by considering the relationship between vertex and edge aggregation quantifiers Q_V and Q_E. Recall that we support quantifiers all, most, at least n, and exists, and observe that they can be translated to a threshold on the percentage of the time during which an entity (vertex or edge) existed, relative to the duration of the aggregation window: $t = 1$ for all, $t > 0.5$ for most, $t > 0$ for exists and $t > n$ for at least n. Let us refer to the vertex threshold as t_V, and to the edge threshold as t_E. If an entity's existence meets the threshold, it will be retained in the result of aggregation.

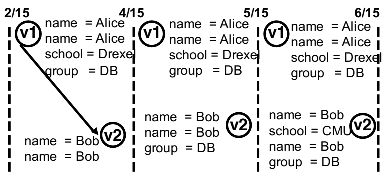
FK enforcement is only required if $t_V > t_E$. To see why, consider an interval w which is one of the windows computed based on specification W. We produce an edge (v_1, v_2, w) iff $\exists\{(v_1, v_2, p_1), \ldots, (v_1, v_2, p_k)\} \in \mathsf{TE} \mid (\bigcup_{i=1}^{k} p_i \cap w)/w > t_E$. According to Definition 5, if the input graph is valid then we must also produce (v_1, w) and (v_2, w) if $t_V \le t_E$, since vertex periods are supersets of edge periods. However, if $t_V > t_E$, we may not produce a vertex that meets t_E but does not meet the greater t_V, as we showed with the example above.

3.5 Temporal graph intersection

The binary temporal graph intersection operation $\mathsf{T_1}^{\mathsf{RG}} \cap \mathsf{T_2}^{\mathsf{RG}}$ computes a temporal join [12] of $\mathsf{T_1}^{\mathsf{RG}}$ and $\mathsf{T_2}^{\mathsf{RG}}$ with the predicate $\mathsf{T_1}^{\mathsf{RG}}.p \cap \mathsf{T_2}^{\mathsf{RG}}.p \neq \emptyset$, producing a tuple for each pair of representative graphs for which time periods intersect: $\mathsf{T_1}^{\mathsf{RG}} \cap \mathsf{T_2}^{\mathsf{RG}} = \{(g_1 \cap g_2, p_1 \cap p_2) \mid ((g_1, p_1) \in \mathsf{T_1}^{\mathsf{RG}} \wedge (g_2, p_2) \in \mathsf{T_2}^{\mathsf{RG}} \wedge p_1 \cap p_2 \neq \emptyset\}$. The result of $g_1 \cap g_2$ is computed by intersecting the sets of vertices and of edges of the graphs [7]. For each vertex and edge in the result, we compute a *union* of their bags of properties.Algorithm 3 presents the evaluation of $\mathsf{T_1}^{\mathsf{VE}} \cap \mathsf{T_2}^{\mathsf{VE}}$. We compute temporal joins over TV and TE (lines 1, 2). We then compute TA^{V}' and TA^{E}' with temporal outer joins of the corresponding relations (lines 3, 4). Finally, we enforce foreign key constraints on TE', TA^{V}' and TA^{E}' (lines 5, 6).


2/15
4/15
6/15
v1 name = Alice group = DB
v1 name = Alice group = DB
name = Bob v2
name = Bob group = DB v2
v4 name = Dave group = AI
v4 name = Dave group = AI

(a) T2.


2/15
4/15
5/15
6/15
v1 name = Alice name = Alice school = Drexel group = DB
v1 name = Alice name = Alice school = Drexel group = DB
v1 name = Alice name = Alice school = Drexel group = DB
name = Bob name = Bob v2
name = Bob name = Bob group = DB v2
name = Bob school = CMU name = Bob group = DB v2

(b) $T1 \cap T2$.

Figure 4: Temporal intersection.

Figure 4 illustrates temporal intersection of TGraph T2, shown in Figure 4a, with T1 in our running example. Period $[2/15, 4/15)$ is computed as a result of the join of $[2/15, 5/15)$ in T1 and $[2/15, 4/15)$ in T2. Only the vertices and edges present in both TGraphs are produced, thus eliminating v_3 and v_4. Notice the result of attribute bag union with duplicates for some keys. These can be resolved by a subsequent map based on the domain requirements.

Intersection may uncoalesce T^{VE} and T^{RG}. While temporal join over coalesced temporal relations does not uncoalesce, as shown in [4], it is followed by a projection, which does uncoalesce. For example, consider a simple TGraph T_2 that consists of only 1 vertex v_1 with attribute (*name* : *Alice*) and no edges and exists for the period $[1/14, 5/15)$. The intersection of T_2 with T in Figure 1 produces the same graph with vertex v_1 for each period of intersection $[1/15, 2/15), [2/15, 5/15)$, which must be coalesced.

Intersection requires FK enforcement for TA^{V} and TA^{E} but not for TE. Consider an edge $\mathsf{TE}(v_1, v_2, p^e) \in \mathsf{TE}'$ and one of the corresponding vertices $\mathsf{TV}(v_1, p^v) \in \mathsf{TV}'$ computed in Algorithm 3. To violate the FK constraint on TE', there must exist a time instant $t \in p^e \wedge t \notin p^v$. But that

Algorithm 3 Temporal graph intersection in T^{VE}.

```
Require: $\mathsf{T_1}^{\mathsf{VE}}, \mathsf{T_2}^{\mathsf{VE}}$.
1: $\mathsf{TV}' = \mathsf{TV}_1 \bowtie_v^T \mathsf{TV}_2$
2: $\mathsf{TE}' = \mathsf{TE}_1 \bowtie_{v_1,v_2}^T \mathsf{TE}_2$
3: $\mathsf{TA}^{\mathsf{V}\prime} = \mathsf{coal}(\pi_{v,p,\mathsf{TA^V}_1.a \cup \mathsf{TA^V}_2.a} \mathsf{TA^V}_1 ⟗_v^T \mathsf{TA^V}_2)$
4: $\mathsf{TA}^{\mathsf{E}\prime} = \mathsf{coal}(\pi_{v_1,v_2,p,\mathsf{TA^E}_1.a \cup \mathsf{TA^E}_2.a} \mathsf{TA^E}_1 ⟗_{v_1,v_2}^T \mathsf{TA^E}_2)$
5: enforce foreign keys on $\mathsf{TA}^{\mathsf{V}\prime}$ w.r.t. $\mathsf{TV}'$
6: enforce foreign keys on $\mathsf{TA}^{\mathsf{E}\prime}$ w.r.t. $\mathsf{TE}'$
7: return new $\mathsf{T}^{\mathsf{VE}}(\mathsf{TV}'; \mathsf{TE}'; \mathsf{TA}^{\mathsf{V}\prime}; \mathsf{TA}^{\mathsf{E}\prime})$
```

Algorithm 4 Temporal graph union in T^{VE}.

```
Require: $\mathsf{T_1}^{\mathsf{VE}}, \mathsf{T_2}^{\mathsf{VE}}$.
 1: $\mathsf{TV}' = \mathsf{TV}_1 ⟗^T_v \mathsf{TV}_2$
 2: $\mathsf{TE}' = \mathsf{TE}_1 ⟗^T_{v1,v2} \mathsf{TE}_2$
 3: $\mathsf{TA}^{\mathsf{V}'} = \mathsf{coal}(\pi_{v,p,\mathsf{TA^V}_1.a \cup \mathsf{TA^V}_2.a} \mathsf{TA^V}_1 ⟗^T_v \mathsf{TA^V}_2)$
 4: $\mathsf{TA}^{\mathsf{E}'} = \mathsf{coal}(\pi_{v_1,v_2,p,\mathsf{TA^E}_1.a \cup \mathsf{TA^E}_2.a} \mathsf{TA^E}_1 ⟗^T_{v1,v2} \mathsf{TA^E}_2)$
 5: return new $\mathsf{T}^{\mathsf{VE}}(\mathsf{TV}'; \mathsf{TE}'; \mathsf{TA}^{\mathsf{V}'}; \mathsf{TA}^{\mathsf{E}'})$
```

is not possible, since by definition of temporal join p^e exists in both TE_1 and TE_2, and $\mathsf{T^{VE}}_1$ and $\mathsf{T^{VE}}_2$ are valid TGraphs. To see why we need to enforce FK for $\mathsf{TA}^{\mathsf{V}'}$, consider again the example above with $T \cap T_2$. The temporal outer join of $\mathsf{TA^V}_1$ and $\mathsf{TA^V}_2$ will produce a tuple for period [1/14, 1/15) with the attribute $(name : Alice)$ from $\mathsf{TA^V}_2$. Clearly this tuple violates the FK constraint because we do not have any vertices in TV' prior to 1/15. Similarly for $\mathsf{TA}^{\mathsf{E}'}$.

3.6 Temporal graph union

The binary temporal graph union operation $\mathsf{T_1}^{\mathsf{RG}} \cup \mathsf{T_2}^{\mathsf{RG}}$ computes a temporal full outer join [12] of $\mathsf{T_1}^{\mathsf{RG}}$ and $\mathsf{T_2}^{\mathsf{RG}}$ on the predicate $\mathsf{T_1}^{\mathsf{RG}}.p \cap \mathsf{T_2}^{\mathsf{RG}}.p$. For tuples $(g_1, p_1) \in \mathsf{T_1}^{\mathsf{RG}}$ and $(g_2, p_2) \in \mathsf{T_2}^{\mathsf{RG}}$ for which $p_1 \cap p_2 \neq \emptyset$, we compute $(g_1 \cup g_2, p_1 \cap p_2)$. The result of $g_1 \cup g_2$ is computed by taking a *union* of the sets of vertices and of edges of the graphs [7]. For each vertex and edge in the result, we compute a *union* of their bags of properties. Tuples from $\mathsf{T_1}^{\mathsf{RG}}$ (resp. $\mathsf{T_2}^{\mathsf{RG}}$) for which there does not exist a tuple in $\mathsf{T_2}^{\mathsf{RG}}$ (resp. $\mathsf{T_1}^{\mathsf{RG}}$) for part or all of the validity period are included in the result of the full outer join. Algorithm 4 presents the evaluation of $\mathsf{T_1}^{\mathsf{VE}} \cup \mathsf{T_2}^{\mathsf{VE}}$. We compute temporal outer joins over the corresponding TV, TE, $\mathsf{TA^V}$ and $\mathsf{TA^E}$.

Union may uncoalesce $\mathsf{T^{VE}}$ and $\mathsf{T^{RG}}$. The logic here is similar to the case with temporal intersection. Because we produce a single graph for each time period by computing a union of g_1 and g_2, a form of projection, the result may be uncoalesced. For example, consider a simple TGraph T_2 that consists of a single graph equal to the first representative graph of T in Figure 1 valid for a period of $[12/14, 5/15)$. The same representative graph will be produced for period $[12/14, 1/15)$ and $[1/15, 2/15)$ and must be coalesced. TV' and TE' are coalesced since they are computed with a temporal join [4]. However, $\mathsf{TA}^{\mathsf{V}'}$ and $\mathsf{TA}^{\mathsf{E}'}$ are computed with a join followed by a projection, which does uncoalesce. Thus we apply the coalescing operation on lines 3 and 4.

Union does not require FK enforcement for $\mathsf{T^{VE}}$. Temporal full outer join produces a vertex tuple for all periods in TV_1 and TV_2. If $\mathsf{T^{VE}}_1$ and $\mathsf{T^{VE}}_2$ are valid graphs, then referential integrity holds for TE', $\mathsf{TA}^{\mathsf{V}'}$ and $\mathsf{TA}^{\mathsf{E}'}$.

3.7 Temporal User-Defined Analytics

For many types of evolving graph analysis, it is necessary to compute some property, such as PageRank of each vertex v, or the length of the shortest path from a given designated vertex u to each v, for each representative graph. This information can then be used to study how the graph evolves over time. Portal supports this type of analysis through *temporal user-defined analytics*, which conceptually execute an analytic over each RG, compute a value for each vertex of the RG, and then store this value among the attributes of the vertex (adding it to the property bag). We provide an API that allows developers to implement custom analytics that can either be computed locally at a vertex, like degree, or that can be expressed in the popular Pregel API [22].

Logically, an analytic is computed on each representative graph. However, we will show in Section 5 that more efficient methods that use batching are possible. Analytics cannot be computed directly on $\mathsf{T^{VE}}$, because they must, by definition, act over representative graphs.

Analytics do not uncoalesce. An analytic adds a new key-value pair to the property bag of each vertex v of every representative graph g, and does not implicitly project out vertex attributes. Thus, an analytic cannot cause two temporally adjacent graphs that were not equivalent before the attribute addition to become equivalent after it was added.

4. SYSTEM

Our Portal system implementation builds on Apache Spark with GraphX [14], an open-source in-memory distributed framework that combines graph parallel and data parallel abstractions. The data is distributed in partitions across the cluster workers, read in from HDFS, and can be viewed both as a graph and as a pair of RDDs. All TGraph operations are available through the public API of the Portal library, and may be used in an Apache Spark application. Portal code is freely available online.

4.1 Lazy Coalescing

All operations and all sequences of operations must output a valid temporally coalesced TGraph. Several implementations are possible for the coalesce operation over temporal SQL relations, see [4] for details. We use the partitioning method, where the relation (e.g., TV, TE, $\mathsf{T^{RG}}$) is grouped by key, and tuples are sorted and folded within each group to produce time periods of maximum length. This involves shuffling between partitions, is computationally expensive, and motivates lazy coalescing.

Correctness of many TGraph algebra operations does not depend on whether their input is coalesced. Consequently, Portal supports both eager and lazy coalescing, with lazy being the default for queries that admit it. We now present useful coalescing rules (T stands for either $\mathsf{T^{RG}}$ or $\mathsf{T^{VE}}$).

Slice allows lazy coalescing. We showed in Section 3.1 that slice does not destroy coalescing. Further, recall that $\tau_c(\mathsf{coal}(\mathsf{T}))$ returns tuples $(x, p \cap c)$, for which $p \cap c \neq \emptyset$. If T is uncoalesced, then there exist some value-equivalent tuples (x, p_1) and (x, p_2) s.t. p_1 **meets** $p_2 \vee p_1$ **contains** $p_2 \vee p_1$ **overlaps** p_2, which would be replaced by $(x, p_1 \cup p_2)$ in $\mathsf{coal}(\mathsf{T})$. Because intersection distributes over union, i.e.,$(p_1 \cup p_2) \cap c = (p_1 \cap c) \cup (p_2 \cap c)$, coalescing can be equivalently performed before or after slice:

$$\mathsf{coal}(\tau_c(\mathsf{T})) \equiv \tau_c(\mathsf{coal}(\mathsf{T})) \tag{10}$$

Temporal subgraph allows lazy coalescing. It was shown in [4] that coalescing can be deferred until after selection if the selection condition is independent of the valid time of the input. Since temporal subgraph (Section 3.2) is a time-invariant form of select, coalescing can be deferred. Subgraph does not uncoalesce $\mathsf{T^{VE}}$ but may uncoalesce $\mathsf{T^{RG}}$ (Section 3.2). For this reason it is required to coalesce the final output in Rule 12 even if the input was eagerly coalesced, but this is not required in Rule 11.

$$\mathsf{coal}(\sigma_{C_V,C_E}(\mathsf{T^{VE}})) \equiv \sigma_{C_V,C_E}(\mathsf{coal}(\mathsf{T^{VE}})) \tag{11}$$

$$\mathsf{coal}(\sigma_{C_V,C_E}(\mathsf{T}^{\mathsf{RG}})) \equiv \mathsf{coal}(\sigma_{C_V,C_E}(\mathsf{coal}(\mathsf{T}^{\mathsf{RG}}))) \qquad (12)$$

Deferring coalescing can be quite effective for Rules 10, 11 and 12 if the selectivity of the predicate is high.

Temporal map allows lazy coalescing, since it processes each input tuple without modifying the corresponding time intervals, and independently of the valid time. Map destroys coalescing (Section 3.3), and requires to coalesce the output even if the input was eagerly coalesced.

$$\mathsf{coal}(\mathsf{map}_{M_V,M_E}(\mathsf{T})) \equiv \mathsf{coal}(\mathsf{map}_{M_V,M_E}(\mathsf{coal}(\mathsf{T}))) \qquad (13)$$

Temporal aggregation requires eager coalescing. When aggregating by change, the input must be coalesced to compute correct aggregation windows. If the input is not coalesced, then there may be two or more consecutive intervals which should be treated as one but would count separately. For aggregation by time, some aggregate functions also require coalesced input. For example, count will produce different result if several consecutive or overlapping value-equivalent tuples fall within the same window. Thus we add the following conditional coalescing rule:

$$\mathsf{coal}(\gamma_{W,Q_V,Q_E,A_V,A_E}(\mathsf{coal}(\mathsf{T}))) \equiv \mathsf{coal}(\gamma_{W,Q_V,Q_E,A_V,A_E}(\mathsf{T}))$$
$$\text{iff } W \text{by\_time} \wedge A_V, A_E \in \{\mathsf{any}, \mathsf{first}, \mathsf{last}, \mathsf{min}, \mathsf{max}\} \qquad (14)$$

any returns any one of the values associated with a property within each group in W, and so is invariant to time and to the number of tuples in W. first and last pick the earliest (resp. latest) value within each group in W. If T is uncoalesced, there is at least one pair of value-equivalent tuples with overlapping or consecutive periods. Selecting between equivalent values will generate the same result. The same argument applies to min and max.

Temporal intersection and union allow lazy coalescing. For T^{RG}, temporal graph intersection (Section 3.5) is a join followed by projection. As shown in [4], coalescing can be deferred until after a join, and after a projection if projection is independent of the valid time interval of the tuple, which is the case here. Thus we can defer coalescing until after temporal graph intersection. Temporal graph union (Section 3.6) is an outer join, and the same argument applies. Note that both operations may destroy coalescing, and so a final coalesce is required over the result even if inputs were eagerly coalesced.

$$\mathsf{coal}(\mathsf{coal}(\mathsf{T}_1) \cap \mathsf{coal}(\mathsf{T}_2)) \equiv \mathsf{coal}(\mathsf{T}_1 \cap \mathsf{T}_2) \qquad (15)$$

$$\mathsf{coal}(\mathsf{coal}(\mathsf{T}_1) \cup \mathsf{coal}(\mathsf{T}_2)) \equiv \mathsf{coal}(\mathsf{T}_1 \cup \mathsf{T}_2) \qquad (16)$$

Temporal user-defined analytics allow lazy coalescing. If T^{RG} is not coalesced, it contains one or more value-equivalent representative graphs with consecutive or overlapping periods. Analytics are time-invariant functions applied to each representative graph. Thus when applied to value-equivalent graphs, they will produce equivalent results, and so coalesce can be deferred. Further, analytics to not destroy coalescing, and so it is not required to coalesce again over the final result.

$$\mathsf{coal}(\mathsf{uda}(\mathsf{T})) \equiv \mathsf{uda}(\mathsf{coal}(\mathsf{T})) \qquad (17)$$

Although it is possible to defer coalesce for analytics, this will not be done in practice. Analytics are computationally expensive, and it is usually beneficial to eagerly coalesce the input, reducing the number of representative graphs.

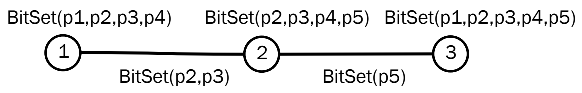


Figure 5: OG representation of T1.

4.2 Physical Representations

We considered four in-memory TGraph representations that differ both in compactness and in the kind of locality they prioritize. With *structural locality*, neighboring vertices (resp. edges) of the same representative graph are laid out together, while with *temporal locality*, consecutive states of the same vertex (resp. edge) are laid out together [23]. We now describe each representation.

RepresentativeGraphs (RG) is a direct implementation of T^{RG} of Definition 2.2. RG is a collection (parallel sequence) of GraphX graphs, where vertices and edges store the attribute values for the specific time interval, thus using structural locality. This representation supports all operations of TGraph algebra. While the RG representation is simple, it is not compact, considering that in many real-world evolving graphs there is a 80% or larger similarity between consecutive snapshots [23]. In a distributed architecture, however, this data structure provides some benefits as operations on it can be easily parallelized by assigning different representative graphs to different workers.

RG is the most immediate way to implement evolving graphs using GraphX. Without Portal a user wishing to analyze evolving graphs might implement and use the RG approach. However, as we will show in Section 5, this would lead to poor performance for most operations.

VertexEdge (VE) is a direct implementation of the T^{VE} model of Definition 2.3, and is the most compact: one RDD contains all vertices and another all edges. Consistently with the GraphX API, all vertex properties are stored together as a single nested attribute, as are all edge properties. We currently do not store the TV and TE relations separately but rather together with TA^{V} and TA^{E}, respectively. While VE does not necessitate a particular order of tuples on disk, we opt for a physical layout in which all tuples corresponding to the same vertex (resp. edge) are laid out consecutively, and so VE preserves temporal locality.

VE supports all TGraph algebra operations but cannot support analytics. This is because an analytic is defined on a representative graph, which VE does not materialize. As we will show in Section 5, due to compactness this physical representation is the most efficient for many operations.

OneGraph (OG) is the most topologically compact representation, which stores all vertices from TA^{V}*and* edges from TA^{E} once, in a single data structure. Information about time validity is stored together with each vertex and edge. Figure 5 shows the OG for T1 from Figure 1. OG emphasizes temporal locality, while also preserving structural locality, but leads to a much denser graph than RG. This, in turn, makes parallelizing computation challenging.

An OG is implemented as a single GraphX graph. To construct an OG from T^{VE}, vertices and edges of TV and TE relations each are grouped by key and mapped to bitsets, which in turn encode the presence of a vertex or edge in each time period associated with some representative graph of a TGraph. Because OG stores information only about graph topology, far fewer periods must be represented and computed for OG than for RG. The actual reduction depends on the rate and nature of graph evolution.

As we will see experimentally in Section 5, OG is often the best-performing data structure for aggregation, and also has competitive performance for analytics. Because of this focus, OG supports operations only on topology: analytics, aggregation. union, and intersection for graphs with no vertex or edge attributes. All other operations are supported through inheritance from an abstract parent, and are carried out on the VE data structure. Thus OG and HG, below, can be thought of as indixes on VE.

HybridGraph (HG) trades compactness of OG for better structural locality of RG, by aggregating together several consecutive representative graphs, computing a single OG for each graph group, and storing these as a parallel sequence. In our current implementation each OG in the sequence corresponds to the same number of temporally adjacent graphs. This is the simplest grouping method, and we observed that placing the same number of graphs into each group often results in unbalanced group sizes. This is because evolving graphs commonly exhibit strong temporal skew, with later graphs being significantly larger than earlier ones. We are currently working on more sophisticated grouping approaches that would lead to better balance, and ultimately to better performance. However as we will see experimentally in Section 5, the current HG implementation already improves performance compared to OG, in some cases significantly.

Like OG, HG focuses on topology-based analysis, and so does not represent vertex and edge attributes. OG implements analytics, aggregation, union, and intersection, and supports all other operations through inheritance from VE.

4.3 Additional Implementation Details

Enforcing referential integrity. The T^{VE} logical representation of a TGraph (Definition 2.3), and the corresponding VE physical representation (Section 4.2), require that referential integrity be maintained by algebraic operations (lines 6—8 of Algorithm 1). We discussed that temporal subgraph (Section 3.2) and aggregation (Section 3.4) both require that tuples be removed from TE, or that their periods of validity be modified to be included within the periods of validity of the corresponding vertices (Condition 5 of Definition 2.3). We do this by executing a join of TE with TV on vertex ids — either a broadcast join if TV is small, or a hash join — and then adjusting time periods as necessary. This is an expensive operation and is only performed when necessary: when temporal subgraph has a non-trivial predicate on C_V, and when temporal aggregation has a more restrictive vertex aggregation quantifier Q_V than edge quantifier Q_E (see Section 3.4 for a discussion).

For the T^{RG} logical representation, and for the physical representations RG, OG and HG, we rely on GraphX to maintain valid representative graphs, and so do not maintain referential integrity explicitly.

Partitioning. Graph partitioning has a tremendous impact on performance. A good partitioning strategy needs to be balanced, assigning an approximately equal number of units to each partition, and limit the number of cuts across partitions, reducing cross-partition communication. In our experiments we compare performance with no repartitioning after load vs. with repartitioning, using the GraphX E2D edge partitioning strategy. In E2D, a sparse edge adjacency matrix is partitioned in two dimensions, guaranteeing a $2\sqrt{n}$ bound on vertex replication, where n is the number of partitions. E2D has been shown to provide good performance for Pregel-style analytics [14, 28].

Graph loading. We use the Apache Parquet format for on-disk storage, with one archive for vertices and another for edges, temporally coalesced. This format corresponds to the VE physical representation 4.2. In cases where there is no more than 1 attribute per vertex and edge, this representation is also the most compact.

For ease of use, we provide a GraphLoader utility that can initialize any of the four physical representations of the TGraph from Apache Parquet files on HDFS or on local disk. The Portal user can also implement custom graph loading methods to load vertices and edges, and then use the fromRDDs to initialize any of the four physical representations.

5. EXPERIMENTAL EVALUATION

Experimental environment. All experiments in this section were conducted on a 16-slave in-house Open Stack cloud, using Linux Ubuntu 14.04 and Spark v2.0. Each node has 4 cores and 16 GB of RAM. Spark Standalone cluster manager and Hadoop 2.6 were used. Because Spark is a lazy evaluation system, a materialize operation was appended to the end of each query, which consisted of the count of nodes and edges. Each experiment was conducted 3 times with a cold start, we report the average running time, which is representative because we took great care to control variability: standard deviation for each measure is at or below 5% of the mean except in cases of very small running times.

OG and HG inherit their implementation of slice, map and subgraph from RG and are not included in these experiments. Performance of all 4 data structures is compared in aggregation, union and intersection.

Data. We evaluate performance of our framework on three real open-source datasets, summarized in Table 1. wiki-talk (`http://dx.doi.org/10.5281/zenodo.49561`) contains over 10 million messaging events among 3 million wiki-en users, aggregated at 1-month resolution.
nGrams (`http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`) contains word co-occurrence pairs, with 30 million word nodes and over 2.5 billion undirected edges. The Twitter social graph [11] contains over 23 billion directed follower relationships between 0.5 billion twitter users, sampled at 1-month resolution. The datasets differ in size, in the number and type of attributes and in evolution rates, calculated as the average graph edit similarity [24].

Slice performance was evaluated by varying the slice time window and materializing the TGraph, and is presented in Figures 6 for wiki-talk and 7 for nGrams. Similar trends were observed for twitter. Slice is expected to be more efficient when executed over VE when data is coalesced on disk than over RG, and we observe this in our experiments. This is because multiple passes over the data are required for RG to compute each representative graph, leading to linear growth in running times for file formats and systems without filter pushdown, as is the case here. Slice over VE simply execute temporal selection and has constant running times (29 sec for wiki-talk, about 1.5 min for nGrams). This experiment essentially measures the cost of materializing RG from a vertex-edge on-disk representation. We observed the same linear trend in our preliminary work, when the data was stored as individual snapshots on disk, although less redundant work is needed in that case.

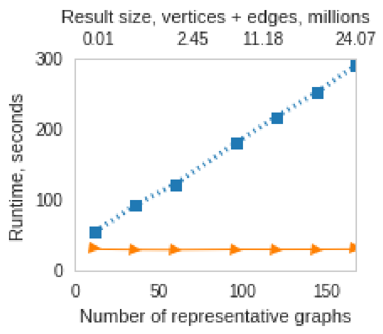


Figure 6: Slice on wiki-talk.

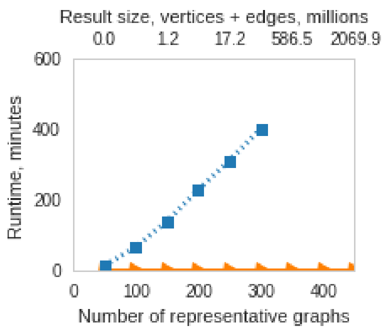


Figure 7: Slice on nGrams.

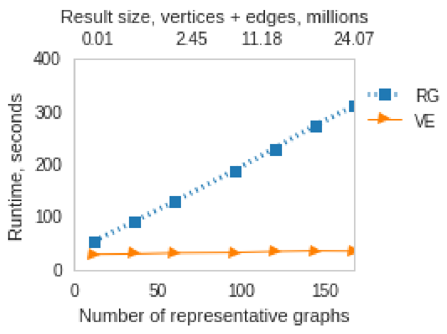
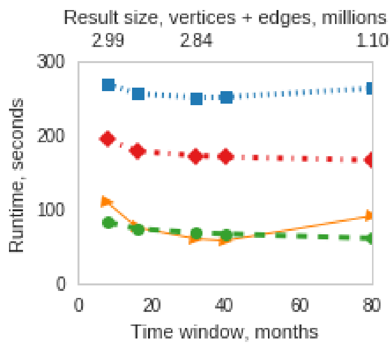
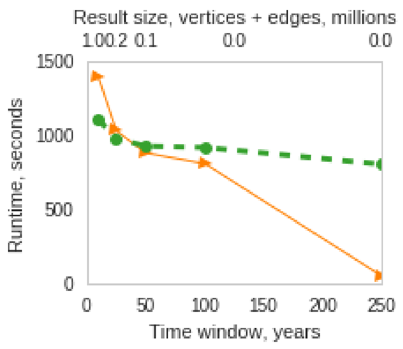


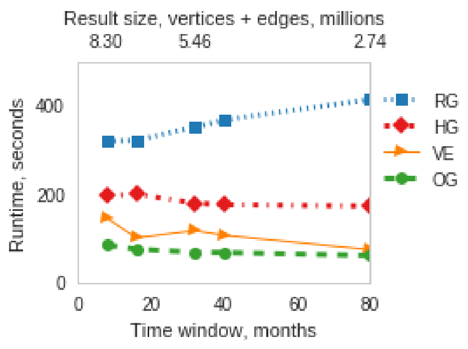
Figure 8: Map on wiki-talk.



(a) $Q_V =$ all, $Q_E =$ all, wiki-talk



(b) $Q_V =$ all, $Q_E =$ exists, nGrams



(c) $Q_V =$ all, $Q_E =$ exists, wiki-talk

Figure 9: Aggregate by time.

Table 1: Experimental datasets.

Dataset	|V|	|E|	Time Span	Evol. Rate
wiki-talk-en	2.9M	10.7M	2002–2015	14.4
nGrams	29.3M	2.5B	1520–2008	16.67
twitter	505.4M	23B	2006–2012	88

Map exhibits a similar trend as slice: constant running time for VE and a linear increase in running time with increasing number of representative graphs for RG (Figure 8 for wiki-talk, similar for other datasets). Performance of map is slightly worse than that of slice because map must coalesce its output as the last step, while slice does not.

Subgraph performance was evaluated by specifying a condition on the $length(a.attr) < t$ of the vertex attribute, with different values of t leading to different selectivity. This experiment was executed for wiki-talk (with *username* as the attribute) and for nGrams (with *word* as the attribute). Twitter has no vertex attributes and was not used in this experiment. Performance on RG is a function of the number of intervals and is insensitive to the selectivity (Figure 13 for nGrams, similar for wiki-talk). The behavior on VE is dominated by FK enforcement: with high selectivity (few vertices) broadcast join affords performance linear in the number of edges, whereas for a large number of vertices broadcast join is infeasible and a hash-join is used instead, which is substantially slower. VE provides an order of magnitude better performance than RG: up to 3 min with hash-join and up to 15 min with broadcast join for VE, in contrast to between 95 and 200 min for RG.

Aggregation performance was evaluated on 4 representations, since all have different implementations of this operator. We executed structure-only aggregation (no attributes), varying the size of the aggregation window. We observed that performance depends heavily on the quantification, and on the data evolution rate. OG is an aggregated data structure with good temporal locality and thus in most cases provides good performance and is insensitive to the aggregation window size (Figure 9a). However, in datasets with a large number of representative graphs (such as nGrams), OG is slow on large windows, an order of magnitude worse than VE (Figure 9b). VE outperforms OG when vertex and edge quantification levels match (Figure 9a), but is worse than OG when vertex quantification is stricter than edge quantification and FK must be enforced (Figure 9c). OG also outperforms VE when both evolution rate is low and aggregation window is small (Figure 9a, wiki-talk).

Union and intersection by structure was evaluated by loading two time slices of the same dataset with varying overlap. Performance depends on the size of the overlap (in the number of representative graphs) and on the evolution rate. VE has best performance when overlap is small (Figure 10). OG always has good performance, constant w.r.t. overlap size. This is expected, since OG union and intersection are implemented as joins (outer or inner) on the vertices and edges of the two operands. VE, on the other hand, splits the coalesced vertices/edges of each of the two operands into intervals first, takes a union, and then reduces by key. When evolution rate is low and duration of an entity is high, such as in wiki-talk for vertices, the split produces a lot of tuples to then reduce, and the performance suffers (Figure 10). RG only has good performance on intersection when few representative graphs overlap, and never on union (Figure 12). HG performance is generally worse than OG, by a constant amount in union, and diverges in intersection.

Analytics. We implemented PageRank (PR) and Connected Components (CC) analytics for the three graph-based representations using the Pregel GraphX API. PR was executed for 10 iterations or until convergence, whichever came first. CC was executed until convergence with no limit on

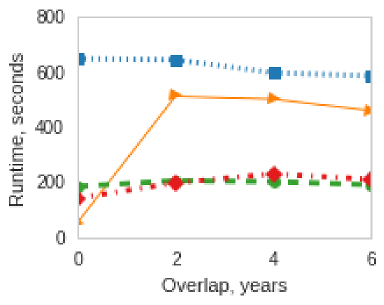


Figure 10: Union on wiki-talk.

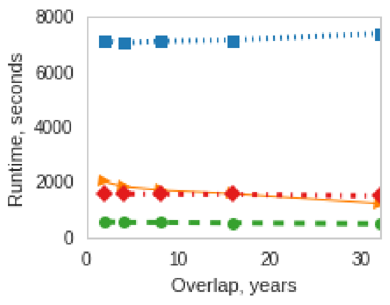


Figure 11: Union on nGrams.

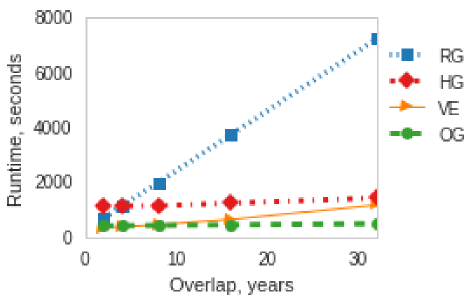


Figure 12: Intersection on nGrams.

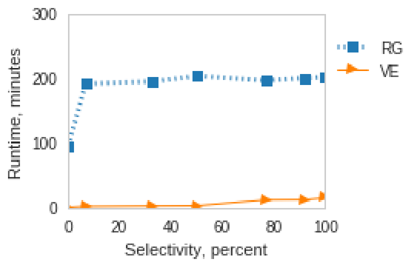


Figure 13: Subgraph on nGrams.

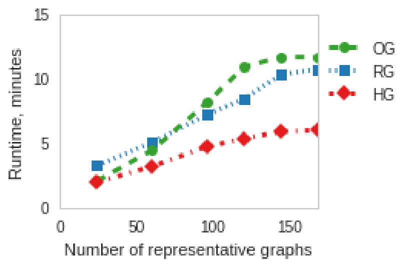


Figure 14: Components on wiki-talk.

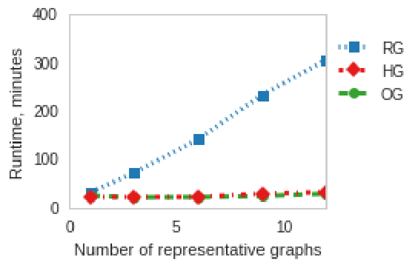


Figure 15: PageRank on Twitter.

Table 2: Effect of cluster size, minutes.

Dataset	4 slaves	8 slaves	12 slaves	16 slaves
wiki-talk	8.41	6.02	5.02	2.94
Twitter	151.77	72.75	55.68	53.46

the number of iterations. Performance of Pregel-based algorithms depends heavily on the partitioning strategy, with best results achieved where cross-partition communication is small [28]. For this reason, we evaluated only with the E2D strategy. Performance was evaluated on time slices of varying size. For a very small number of graphs (1-2), RG provides good performance, but slows down linearly as the number of graphs increases. HG provides the best performance on analytics under most conditions, with a linear increase but a significantly slower rate of growth. The tradeoff between OG and HG depends on graph evolution characteristics. If the graph is a growth-only evolution (such as in Twitter), OG is not denser than HG and computes everything in a single batch, which leads to the fastest performance, as can be seen in Figure 15. If the edge evolution represents more transient connections, then HG is less dense and scales better (Figure 14). Note that OG and HG performance could be further improved by computing them over coalesced structure-only V and E, and ignoring attributes.

Cluster Size. We next examine how the system performance scales with the size of the cluster. We execute slice; aggregate(1 year, exists, exists); components. This query, executed on the wiki-talk dataset, computes connected components over the past 10 years on the yearly scale. Over the twitter dataset the slice size is 2 years.

The best performing representation was selected based on our experimental results. Slice, project, and aggregate were performed with VE, analytics with HG. The results are in Table 2. Both queries show improvements in running time as the cluster size grows, with diminishing returns.

In summary, no one data structure is most efficient across all operations. VE provides the best performance for slice, map, and subgraph. OG is efficient for aggregation under most conditions, and HG for analytics. The graph evolution rate is an important factor in selecting the more suitable representation.

6. RELATED WORK

Temporal data models and languages. The TGraph representation is based on interval semantics [16], and is temporally coalesced [4]. TGraph algebra is based on the principle of snapshot reducibility [5]. We use insights of temporal joins [12] to define TGraph union and intersection. aggregation is inspired by stream aggregation of [21]. An interval-based model of graph evolution, and a composable algebra for evolving graphs based on snapshot reducibility, are among our most important contributions.

Evolving graph models and representations. Much recent work represents evolving graphs as sequences of snapshots in a discrete time domain, and focuses on snaposhot retrieval and analytics [18, 23, 24]. This approach is in contrast to an interval-based model that we use and has several limitations. First, the state of the graph can be undefined at some time t unless a snapshot is associated with each possible value in the discrete range $[t_{start}, now)$. Second, it is not compact: every change to an entity (vertex or edge) requires a new snapshot. The G* system [20] manages graphs that correspond to periodic snapshots, with the focus on efficient data layout. It takes advantage of the similarity between successive snapshots by storing shared vertices only once and maintaining per-graph indexes.

Efficient physical representations using deltas are investigted in [18]. Their on-disk representation is compatible with the period-based model, but the logical model and the algorithms for retrieval are snapshot-based. Their in-memory GraphPool maintains a single representation of all snapshots, and stores only dependencies from a materialized snapshot when deltas are small. Evaluation of queries involving multiple snapshots, such as aggregation, requires fully materialized views in memory, which makes this approach infeasible for our purposes.

Ren et al. [24] develop an in-memory representation of evolving graphs based on representative graphs for sets of snapshots. Their notion of a representative graph differs from ours, in that it indicates a union or an intersection between a set of consecutive snapshots rather than being a consistent representation of the graph for some period p. Nevertheless, Ren's representative graphs can be computed in Portal using temporal aggregation.

Semertzidis et al. [26] develop a version graph, where each node and edge is annotated with the set of time intervals in which they exist. Their model is also a sequence of snapshots with the integer time domain. The version graph can be extended to support the period-based model, and is similar to our OG repesentation. Boldi et al. [6] present a space-efficient non-delta approach for storing a large evolving Web graph that they harvested, representing purely topological information, with no vertex or edge attributes.

Another commonly used graph model is the continuous time model based on change streams [9, 10], usually to support analysis of the latest state of the graph. Here, stream S emits a sequence of graph change events e, each associated with a time t at which it was emitted. A TGraph can be constructed from a change stream following the conventional temporal database approach of maintaining valid-time data.

Querying and analytics. There has been much recent work on analytics for evolving graphs, see [1] for a survey. This line of work is synergistic with ours, since our aim is to provide systematic support for scalable querying and analysis of evolving graphs.

Several researchers have proposed individual queries, or classes of queries, for evolving graphs, but without a unifying syntax or general framework. Kan et al. [17] propose a query model for discovering subgraphs that match a specific spatio-temporal pattern. Chan et al. [8] query evolving graphs for patterns represented by waveforms. Semertzidis et al. [26] focus on historical reachability queries.

Miao et al. [23] developed ImmortalGraph, a proprietary in-memory execution engine for temporal graph analytics. Their approach is based on the snapshot model and does not provide a query language.Nonetheless, many insights about temporal vs. structural locality by [23] hold in our setting.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented an interval-based model of evolving graphs, and proposed a composable TGraph algebra that supports a rich set of operations. It is in our immediate plans to develop a declarative syntax for the algebra, making it accessible to a wider audience of users.

We described an impementation in scope of Apache Spark, and studied performance of operations on different physical representations. Interestingly, different physical implementations perform best for different operations, opening up avenues for rule-based and cost-based optimization. Developing a query optimizer for Portal is in our immediate plans.

8. REFERENCES

[1] C. C. Aggarwal and K. Subbian. Evolutionary network analysis. *ACM Comput. Surv.*, 47(1):10:1–10:36, 2014.

[2] J. Allen. Maintaining knowledge about temporal intervals. *CACM*, 26(11), 1983.

[3] A. Beyer et al. Mechanistic insights into metabolic disturbance during type-2 diabetes and obesity using qualitative networks. *T. Comp. Sys. Biology*, 12, 2010.

[4] M. H. Böhlen et al. Coalescing in temporal databases. In *VLDB*, 1996.

[5] M. H. Böhlen et al. Temporal compatibility. In *Encyclopedia of Database Systems.* 2009.

[6] P. Boldi et al. A Large Time-Aware Web Graph. *ACM SIGIR Forum*, 42(2), 2008.

[7] J. Bondy and U. Murty. *Graph theory.* Springer, 2008.

[8] J. Chan et al. Discovering correlated spatio-temporal changes in evolving graphs. *KAIS*, 16(1), 2008.

[9] R. Cheng et al. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys*, 2012.

[10] D. Ediger et al. STINGER: High performance data structure for streaming graphs. In *HPEC*, 2012.

[11] M. Gabielkov et al. Studying social networks at scale: Macroscopic anatomy of the twitter social graph. In *SIGMETRICS*, 2014.

[12] D. Gao et al. Join operations in temporal databases. *VLDB Journal*, 14(1), 2005.

[13] J. Gonzalez et al. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[14] J. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[15] P.-y. Hsu and D. S. Parker. Improving SQL with Generalized Quantifiers. In *ICDE*, 1995.

[16] C. S. Jensen and R. T. Snodgrass. Temporal data models. In *Encyclopedia of Database Systems.* 2009.

[17] A. Kan et al. A Query Based Approach for Mining Evolving Graphs. In *AusDM*, volume 101, 2009.

[18] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In *ICDE*, 2013.

[19] K. G. Kulkarni and J. Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[20] A. G. Labouseur et al. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 33(4), 2015.

[21] J. Li et al. Semantics and evaluation techniques for window aggregates in data streams. In *ACM SIGMOD*, 2005.

[22] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, 2010.

[23] Y. Miao et al. ImmortalGraph: A system for storage and analysis of temporal graphs. *TOS*, 11(3):14, 2015.

[24] C. Ren et al. On Querying Historical Evolving Graph Sequences. *PVLDB*, 4(11), 2011.

[25] I. Robinson, J. Webber, and E. Eifrem. *Graph databases.* O'Reilly Media, Inc., 2013.

[26] K. Semertzidis et al. TimeReach: Historical reachability queries on evolving graphs. In *EDBT*, 2015.

[27] L. Yang et al. Link analysis using time series of web graphs. In *CIKM*, 2007.

[28] V. Zaychik Moffitt and J. Stoyanovich. Towards a distributed infrastructure for evolving graph analytics. In *TempWeb*, 2016.

[29] E. Zimányi. Temporal aggregates and temporal universal quantification in standard SQL. *SIGMOD Record*, 35(2):16–21, 2006.