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?

Much research and engineering effort today goes into developing sophisticated graph analytics and their efficient implementations, both stand-alone and in scope of data processing platforms. Yet, *systematic support* for scalable querying and analytics over *evolving graphs* still lacks.

In this paper we present Portal, a declarative language that supports efficient querying and exploratory analysis of evolving graphs, and an implementation of Portal in scope of Apache Spark, an open-source distributed data processing framework. Our language supports a variety of operations including temporal selection, join and aggregation, and a rich class of analytics. We develop multiple physical representations and partitioning strategies and study the trade-offs between structural and temporal locality. We provide an extensive experimental evaluation of our system, demonstrating that careful engineering can lead to good performance.

1. INTRODUCTION

The importance of networks in scientific and commercial domains cannot be overstated. Networks are represented by graphs, and we will use the terms “network” and “graph” interchangeably. Considerable research and engineering effort is being devoted to developing effective and efficient graph representations and analytics. As of late, efficient graph abstractions and analytics for *static graphs* have become available in scope of open source platforms such as Apache Giraph [8], Apache Spark, through the GraphX API [12], and GraphLab, through the PowerGraph library [10]. This in turn makes sophisticated graph analysis methods available to researchers and practitioners, facilitating their widespread adoption. Further, because these systems are open source, this encourages development and dissemination of new graph analysis methods, and of more efficient implementations of existing methods.

Arguably the most interesting and important questions one can ask about networks have to do with their evolution, rather than with their static state. Analysis of *evolving graphs* has been receiving increasing attention, with most progress taking place in the last decade [2, 6, 14, 22, 26, 28]. Some areas where evolving graphs are being studied are social network analysis [9, 18, 19, 27], biological networks [3, 4, 29] and the Web [7, 25].

Yet, despite much recent interest and activity on the topic, 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 declarative query language, and its implementation in an open-source distributed framework, that fills this gap.*

To further motivate our work, let us consider several categories of questions one may ask about evolving networks, as part of exploratory analysis.

Which network nodes are showing an increasing popularity trend, or have increasing influence, and which are on a downward spiral? In the Web graph this information can help prioritize crawling. In social networks – for content recommendation and advertisement targeting. In semantic knowledge bases – to capture the dynamics of zeitgeist. Here, node popularity can be quantified in a number of ways including, e.g., node degree, centrality, or PageRank score. Portal supports efficient computation of node popularity with *snapshot and trend analytics.*

Have any changes in network connectivity been observed, either suddenly or gradually over time? For networks describing insulin-based metabolism pathways, gradual pathway disruption can be used to determine the onset of type-2 diabetes [4]. For a website accessibility network, sudden loss of connectivity can signal that censorship is taking place, e.g., in response to a recent election or another exogenous event. In a co-authorship network, increasing connectivity among topical communities indicates stronger collaboration across domains. Here, again, connectivity can be quantified as, e.g., pair-wise distance, length of shortest path between communities, or graph density. Portal supports this kind of analysis with *snapshot and trend analytics.*

At what time scale can interesting trends be observed? The answer to this question may not be known apriori, at the time when graph evolution data is being recorded. For example, changes in node centrality in a social network may be observable on the scale of weeks, but not months (coarser). On the Web, periodic events may change popularity of websites, with observable trends on the scale of days, but not hours (finer) or months (coarser). Furthermore, the same network may show different kinds of trends at different time scales, e.g., node popularity may be changing at a different rate, and thus be observable on a different scale, than over-all network density. Understanding at what temporal scale to consider network evolution is an integral part of exploratory analysis. This exploratory analysis is supported in Portal with *temporal aggregation*.

Can information about graph evolution be used to make graph analytics more stable, or representative? Algorithms that compute website popularly can be vulnerable to link spam, which is a persistent phenomenon on the Web, but the identity of spammers is transient [32]. This suggests that persistence vs. transience of a node, edge, or, more generally, of a subgraph, is a meaningful aspect of quality. Stable or representative subgraphs have also been used to improve performance of iterative computations in evolving graphs, e.g., for computing shortest paths [26]. The *temporal aggregation* operation in Portal can be used to find a representative subgraph of an evolving graph.

How can multiple data sources be used jointly, to complement or corroborate information about graph evolution? It may be the case that multiple datasets are 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 even at different temporal scales. Can these datasets be unified, in a principled way, to support analysis or meta-analysis of network evolution trends? Portal supports several variants of the *temporal join operator* to enable this kind of analysis.

Contributions. In this paper we develop Portal, a declarative query language for efficient querying and exploratory analysis of evolving graphs. Our language supports a variety of operations including temporal selection, join, and aggregation, and a rich class of analytics. Further, we provide a scalable and extensible open-source implementation of Portal in scope of Apache Spark, an open-source distributed data processing framework. We develop several novel physical representations of evolving graphs, and novel partitioning strategies that explore the trade-off between structural and temporal locality. We experimentally demonstrate that good performance can be achieved with careful engineering.

Roadmap. We present our model in Section 2. We describe the Portal query language in Section 3, and discuss its implementation in an open-source prototype in Section 4. Section 5 describes our extensive experimental evaluation on real datasets. We discuss related work in Section 6. Future work and conclusions are given in Section 7.

2. MODEL

The basic element of our model is a TGraph, which represents a graph that evolves over time. We define snapshots of an evolving graph in Section 2.1. We then formalize the temporal aspect of our model in Section 2.2. Finally, in Section 2.3 we show how graph evolution is represented by assigning temporal meaning to sequences of snapshots.

2.1 Snapshots

A state of an evolving graph is represented by a snapshot.

Definition 2.1 (Snapshot). *A* snapshot *G is a pair $(V;E)$, where V is a finite set of nodes with schema $(\underline{vid}, a_1, \ldots, a_n)$, and E is a finite set of edges connecting pairs of nodes from V, with schema $(\underline{vid_1}, \underline{vid_2}, b_1, \ldots, b_m)$.*

Attributes of vertices and of edges are not restricted to be of atomic types, but may, e.g., be maps or tuples. However, we require that all vertices (resp. edges) of G have the same schema, i.e., V and E are homogeneous sets. G may represent a directed or an undirected graph. For undirected graphs we choose a canonical representation of an edge, with $vid_1 \leq vid_2$ (self-loops are allowed).

Definition 2.2 (Structural union-compatibility). *Snapshots $G' = (V', E')$ and $G'' = (V'', E'')$ are union-compatible if V' and V'' are union-compatible, and E' and E'' are union-compatible.*

Several operations of our query language combine two or more snapshots into a single snapshot. This is done using structural aggregation, defined next.

Definition 2.3 (Structural aggregation). *A pair of operators $(\gamma^V, \gamma^E) : \mathcal{G} \rightarrow G$ map a collection of snapshots $\mathcal{G}$ to a snapshot G. γ^V and γ^E specify aggregation behavior for the vertices and for the edges, respectively. For each, we support two variants. (Here, L refers to the list of aggregated attributes.)*

[Any V] $V(G) = \gamma^V_{vid,L}(\bigcup_{G_i \in \mathcal{G}} V(G_i))$. Vertices in the resulting snapshot, G, are present in any snapshot *in $\mathcal{G}$.*

[All V] $V(G) = \gamma^V_{vid,L}(\bigcap_{G_i \in \mathcal{G}} V(G_i))$. Vertices in G are present in all snapshots *in $\mathcal{G}$.*

[Any E] $E(G) = \gamma^E_{vid_1,vid_2,L}(\bigcup_{G_i \in \mathcal{G}} E(G_i))$. Edges in G are in any snapshot *in $\mathcal{G}$, subject to $vid_1, vid_2 \in V(G)$.*

[All E] $E(G) = \gamma^E_{vid_1,vid_2,L}(\bigcap_{G_i \in \mathcal{G}} E(G_i))$. Edges in G are in all snapshots *in $\mathcal{G}$, subject to $vid_1, vid_2 \in V(G)$.*

Note that both γ^V and γ^E group the respective collections by their key attributes. The choice of union (Any) vs. intersection (All) for vertices and for edges is orthogonal, subject to the constraint that an edge is present in the result only if both vertices it connects are present. Finally, note that we did not specify (i) whether $\mathcal{G}$ is an ordered collection of snapshots, and (ii) which aggregation operations are available for non-key attributes of vertices and edges. We will address both points in Section 3, when we discuss query language operations that use structural aggregation.

2.2 Temporal sequences

We next describe how time is represented in our model. Following the SQL:2011 standard [17], we adopt the *closed-open* period model, where a period represents all times starting from and including the start time, continuing to but excluding the end time.

Definition 2.4 (Time period). *A* time period *$p = [start, end)$ is an interval on the timeline, subject to the constraint $start < end$. We refer to the length of time covered by p as its* resolution*, which is explicitly associated with a* unit of time.

Examples of time periods are $p_1 = [2000, 2001)$ (a 1-year period), $p_2 = [2000/12, 2001/03)$ (a 3-month period) and $p_3 = [2000/12/01, 2001/03/01)$ (a 90-day period). The unit of time is application-dependent, and will be stated explicitly where not clear from context.

Our goal in this work is to support complex analytics over evolving graphs, under the assumption that all historical data is available in the database and is read-only. For this reason we focus on *valid time*, represented by *application-time period* in SQL:2011 — the time period during which data is regarded as correctly reflecting reality. This is in contrast to *transaction time* (or *system-time period*), which refers to the time period during which a row is committed to the database.

We now formally define temporal sequences.

Definition 2.5 (Temporal Sequence). *A* temporal sequence $P = (p_1, \ldots, p_n)$ *is a sequence of consecutive non-overlapping time periods of the same resolution, with no gaps. That is,*

1. $\forall i < n, p_i.end = p_{i+1}.start$, *and*
2. $\forall i, j \leq n, p_i.end - p_i.start = p_j.end - p_j.start$.

P may be equivalently described by any two of the following three values: the start of the earliest period $P.start = p_1.start$, the end of the latest period $P.end = p_n.end$, and the resolution of any period $P.res = p_1.end - p_1.start$, specified in appropriate time units. For convenience, we refer to the number of periods in the sequence as $P.size$.

For example, $P = ([1940, 1945), \ldots, [2010, 2015))$ represents a temporal sequence with $P.start = 1940$, $P.end = 2015$, $P.res = 5$ years, and $P.size = 15$.

A special sequence P^ϵ is the null sequence, with $P^\epsilon.res = null$, $P^\epsilon.start = null$, $P^\epsilon.end = null$, and $P^\epsilon.size = 0$.

We next define union-compatibility for temporal sequences, and present two binary operations on temporal sequences.

Definition 2.6 (Temporal Union-Compatibility). *Temporal sequences P' and P'' are union-compatible if they have the same resolution, and if we can construct a valid sequence P with $P.start = min(P'.start, P''.start)$, $P.end = max(P'.end, P''.end)$, and $P.res = P'.res$. P^ϵ is union-compatible with any temporal sequence.*

Example 1. *Consider the following temporal sequences, with year as the unit of time.*

$$\begin{aligned} P_1 &= ([2001, 2003), [2003, 2005)) \\ P_2 &= ([2009, 2010), [2010, 2011)) \\ P_3 &= ([2008, 2010), [2010, 2012), [2012, 2014)) \\ P_4 &= ([2012, 2014), [2014, 2016)) \\ P_5 &= ([2020, 2022)) \end{aligned}$$

P_1 and P_2 are not union-compatible because the resolution of P_1 is 2 years, while the resolution of P_2 is 1 year. P_1 and P_3 are not union-compatible because, while their resolution is the same, it is not possible to construct a valid temporal sequence with $P.start = 2001$, $P.end = 2014$ and a 2-year resolution. Finally, P_3, P_4 and P_5 are pair-wise union-compatible.

As our example illustrates, a pair of union-compatible sequences need not overlap and need not be consecutive.

Definition 2.7 (Temporal Intersection). *Intersection of union-compatible sequences P' and P'', denoted $P' \cap P''$, is a sequence P, containing intervals that are in common to P' and P''. If no intervals are in common to P' and P'', this operation returns P^ϵ.*

Continuing with Example 1, $P_3 \cap P_4 = ([2012, 2014))$ and $P_3 \cap P_5 = P^\epsilon$.

Definition 2.8 (Temporal Union). *Temporal union of union-compatible sequences P' and P'', denoted $P' \cup P''$, is the sequence P with $P.start = min(P'.start, P''.start)$, $P.end = max(P'.end, P''.end)$, and $P.res = P'.res$.*

In Example 1, $P_3 \cup P_4 = ([2008, 2010), \ldots, [2014, 2016))$ and $P_3 \cup P_5 = ([2008, 2010), \ldots, [2020, 2022))$.

Finally, we define temporal aggregation, a coarsening operation for sequences.

Definition 2.9 (Temporal Aggregation). *Operator $\gamma_w(P)$, with $w \geq P.res$, returns a temporal sequence P' such that $P.start = P'.start$, $P.end = P'.end$ and $P'.res = w$, or P^ϵ if a valid sequence cannot be constructed.*

For example, $\gamma_{4y}(P1) = ((2001, 2005])$, and $\gamma_{3y}(P1) = P^\epsilon$. Temporal aggregation is semantic, e.g., an input sequence that has day as its unit, and 1-day resolution, can be coarsened into a sequence with month as its time unit, by specifying $w = 1$ month.

2.3 TGraphs

A snapshot represents a single state of an evolving graph, and is not time-aware. Temporal evolution of a graph is represented by a sequence of snapshots, called a *temporal graph*, or TGraph for short.

Definition 2.10 (TGraph). *A* temporal graph $T = (G_1, \ldots, G_n; P)$ *associates a sequence of n structurally union-compatible snapshots with a temporal sequence P, such that $P.size = n$.*

Snapshots define the *structural schema* of T, while P specifies the *temporal schema* of T. An example of a TGraph is given in Figure 1, with instances of vertex and edge relations for 3 snapshots in Figure 3. Importantly, identity of a vertex persists across snapshots in a TGraph, and across TGraphs. For example, vertex with id 3 represents the same entity in all snapshots in Figure 1 in which it occurs.

To conclude this section, we define union-compatibility for TGraphs.

Definition 2.11 (TGraph Union-Compatibility). *T' and T'' are union-compatible TGraphs if they are both structurally union-compatible (per Definition 2.2) and temporally union-compatible (per Definition 2.6).*

The TGraph of Definition 2.10 is the basic element in our model. In what follows, we assume that a relation in our database corresponds to a single TGraph, not to a collection of TGraphs. In the next section we will present the Portal query language that operates on TGraphs.

3. PORTAL BY EXAMPLE

In this section we present Portal, a declarative query language for evolving graphs. Portal uses SQL-like syntax, and has the form TSelect ... From ... TWhere ... TGroup. We prefix temporal keywords with T, to make the distinction between Portal and SQL operations explicit. We will use TGraphs T1 and T2 of Figures 1 and 2, with structural schema V(<u>vid</u>:int, name:str, salary:int); E(<u>vid1</u>:int, <u>vid2</u>:int, cnt:int), in our examples.

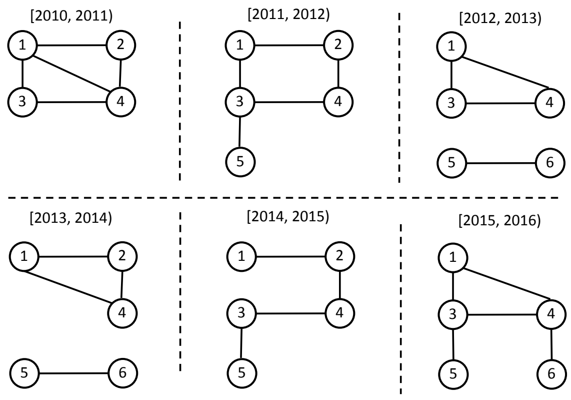


Figure 1: TGraph T1 with 6 snapshots.

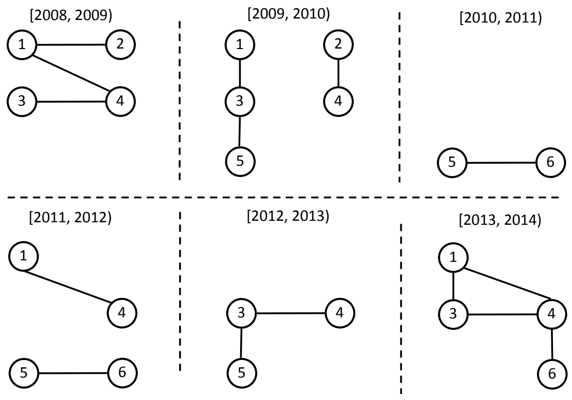
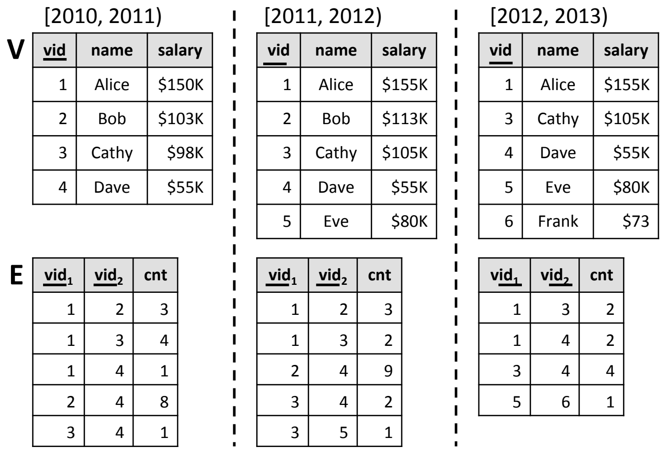


Figure 2: TGraph T2 with 6 snapshots.



V, [2010, 2011)

vid	name	salary
1	Alice	\$150K
2	Bob	\$103K
3	Cathy	\$98K
4	Dave	\$55K

V, [2011, 2012)

vid	name	salary
1	Alice	\$155K
2	Bob	\$113K
3	Cathy	\$105K
4	Dave	\$55K
5	Eve	\$80K

V, [2012, 2013)

vid	name	salary
1	Alice	\$155K
3	Cathy	\$105K
4	Dave	\$55K
5	Eve	\$80K
6	Frank	\$73

E, [2010, 2011)

vid_1	vid_2	cnt
1	2	3
1	3	4
1	4	1
2	4	8
3	4	1

E, [2011, 2012)

vid_1	vid_2	cnt
1	2	3
1	3	2
2	4	9
3	4	2
3	5	1

E, [2012, 2013)

vid_1	vid_2	cnt
1	3	2
1	4	2
3	4	4
5	6	1

Figure 3: Vertices and edges of 3 snapshots of T1.

3.1 Portal Basics

Temporal selection. Consider query $Q1$ below.

```
Q1: TSelect  V; E
    From     T1
    TWhere   Start >= 2010 And End < 2014
```

$Q1$ performs temporal selection — its result is a TGraph that contains a consecutive subset of the snapshots of T1 that fall within the interval specified by the TWhere clause, namely, $[2010, 2011)$ through $[2013, 2014)$. The result of $Q1$ has the same structural schema as T1. The Start or the End portions of the TWhere clause may be omitted.

More generally, the TWhere clause supports a variety of predicates based on which it is determined, for each snapshot in a TGraph, whether that snapshot is to be retained or discarded. For example, TWhere month(Start) like '%r%' will retain all snapshots that start during a month in which it is safe to eat oysters (these are months with the letter 'r' in their name), while TWhere year(End) % 4 = 0 will retain snapshots that end in a leap year, and discard all others. However, recall from Definition 2.5 that a temporal sequence, which constitutes the temporal schema of a TGraph, cannot have any gaps. We enforce this by never discarding a time interval in the middle of a sequence, but rather replacing its snapshot with an empty graph $G(V = \emptyset; E = \emptyset)$.

Specifying structural schema of the result. Snapshot analytics. Next, consider query $Q2$ below.

```
Q2: TSelect  V [vid, pagerank() as pr];
             E [vid1, vid2, cnt * 0.001 as score]
    From     T1
```

This query illustrates how the TSelect clause can be used to specify the structural schema of the result, which in this case is V(<u>vid</u>:int, pr:float); E(<u>vid1</u>:int, <u>vid2</u>:int, score:float). We may use the TSelect clause to project out non-key columns of V and E, or to add columns with computed values. Data types of computed attributes pr and score are determined by the return type of the expressions that compute them. Note that key columns of V and E must be present in the result.

The value of the attribute pr in $Q2$ is computed using a snapshot analytic function pagerank(). Portal supports a variety of snapshot analytics — functions whose values are computed w.r.t. each snapshot of a TGraph— including degree, shortest paths, and connected components. We provide an API that allows developers to implement custom analytics that can either be computed locally at a vertex, like degree, or be expressed in the popular Pregel API [21].

3.2 Temporal Aggregation and Join

Temporal aggregation is illustrated by query $Q3$, which, when executed with T1 from Figure 1 as input, computes the TGraph in Figure 4.

```
Q3: TSelect  Any V ; Any E
    From     T1
    TGroup   by 2 years
```

Temporal aggregation is a two-step operation. First, temporal schema of the output is computed according to Definition 2.9. Then structural aggregation of Definition 2.3 is used over snapshots in the same temporal group. Note the use of Any V and Any E in the TSelect clause of $Q3$, specifying that γ^V and γ^E operate over unions of vertices and edges. For an example consider snapshot $[2010, 2012)$ in Figure 4, which is computed from snapshots $[2010, 2011)$ and $[2011, 2012)$ of T1 in Figure 1.

We may use TGroup by Size to specify that all snapshots of the input TGraph be aggregated into a 1-snapshot TGraph. This notation will be useful when we discuss trend analytics later in this section.

Consider next query $Q4$, and its result in Figure 5.

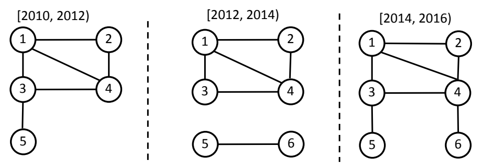


Figure 4: Result of query Q3 on T1.

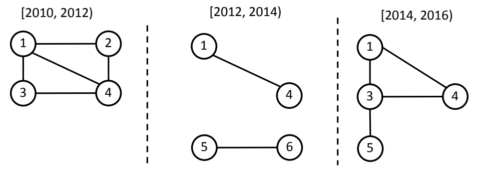


Figure 5: Result of query Q4 on T1.

```
Q4: TSelect All V [vid, any(name), max(salary)] ;
            Any E [vid1, vid2, sum(cnt)]
    From T1
    TGroup by 2 years
```

The main difference between $Q4$ and $Q3$ is the All modifier associated with vertices in the TSelect clause of $Q4$, meaning that $\gamma^V_{vid,any(name),max(salary)}(\bigcap_{G_i \in \mathcal{G}} V(G_i))$ (Definition 2.3) is used to aggregate vertices of snapshots in each 2-year group. Any E states that the edges in the result correspond to the union of the edges connecting the vertices.

$Q4$ illustrates another important feature of Portal, namely, aggregation of values of non-key attributes of vertices and edges, which takes place as part of structural aggregation. We left available operations unspecified in Definition 2.3, so as to keep structural aggregation generic. When used in scope of TGroup, structural aggregation operates over an ordered collection of snapshots. Portal makes the following aggregation operations available for ordered snapshot collections: any, first, last, min, max, sum, count, and list.

To illustrate, consider vertex and edge relations in Figure 3, which correspond to the first three snapshots of T1. Vertex 1 is present in both $[2010, 2011)$ and $[2011, 2012)$ in T1, and so is present in the snapshot $[2010, 2012)$ of the result of $Q4$. Vertex 1 has name='Alice' in both snapshots, but different values for salary. Therefore, taking any value of name and the maximum salary may be appropriate. Operations first and last return the value corresponding to the earliest (resp. latest) occurrence of an attribute, while list returns a collection of all attribute values.

Returning to query $Q3$, when aggregation of attribute values is not specified explicitly, any is used as the default for non-key attributes. That is, the TSelect clause of $Q3$ is short-hand for TSelect Any V[vid, any(name), any(salary)] ; Any E[vid1, vid2, any(cnt)].

Temporal join. We will now present two binary operators of Portal that join together TGraph relations, and will illustrate them using T1 (Figure 1) and T2 (Figure 2). Query $Q5$ computes temporal intersection of T1 and T2. We require that T1 and T2 be union-compatible, as per Definition 2.11. The result of this query is in Figure 6.

```
Q5:  TSelect   Any V; Any E
     From      T1 TAnd T2
```

Temporal schema of the result is computed according to Definition 2.7, and corresponds to a sequence with $P.start =$ 2010, $P.end = 2014$ and $P.size = 4$. Note the use of Any V and Any E in the TSelect clause. This is another example of structural aggregation (Definition 2.3), now as part of TAnd. The modifiers Any and All have the same meaning here as for TGroup, specifying that, a vertex (resp. edge) will be present in a snapshot in the result if it is present in at least one corresponding snapshot of T1 or T2.

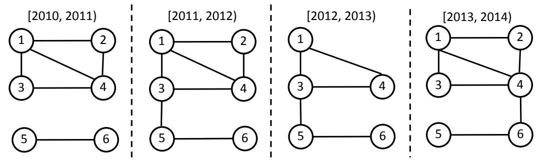


Figure 6: Result of query Q5.

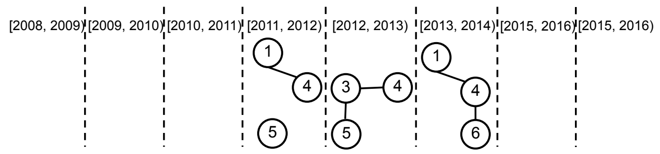


Figure 7: Result of query Q6.

An important difference is that, for TAnd and TOr, structural aggregation operates on unordered collections of snapshots, with at most two snapshots per group. Portal makes the following aggregation operations available for unordered snapshot collections: any, min, max, sum, count, and list. Note that, unlike for TGroup, first and last are unavailable here, and that any is still the default.

Consider next query $Q6$ that computes temporal union of T1 and T2. The result of this query is shown in Figure 7.

```
Q6:  TSelect    All V[vid, pagerank() as pr];
                Any E[vid1, vid2, sum(cnt)]
     From       T1 TOr T2
```

Temporal schema of the result is computed according to Definition 2.8, and corresponds to a sequence with $P.start =$ 2008, $P.end = 2016$ and $P.size = 8$. Further, observe the use of projection (non-key attributes of V are not retained), of an analytic function pagerank(), and of the aggregation operation sum applied to the edge attribute cnt.

3.3 Complex queries

Order of operators. So far we illustrated individual operators of Portal. Now we will show how multiple operators can be combined in a single query. The logical order of evaluation of a Portal query without nesting is as follows:

1. Temporal selection in the TWhere clause;
2. Temporal join (TAnd and TOr) in the From clause;
3. Temporal aggregation in the TGroup clause;
4. Projection, computation of attribute aggregates and analytic functions.

While the logical order of operations is predetermined, we will see in Section 4.1 that some operators can be reordered without affecting the result (but with potential differences in performance), while others cannot.

Consider $Q7$ below, with result shown in Figure 8. $Q7$ first executes temporal selection on each T1 and T2. (Note that when temporal conditions in the TWhere clause are not qualified, they are applied to all TGraphs in the From clause.) Next, $Q7$ computes temporal intersection of T1 and T2, and

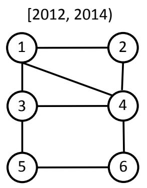


Figure 8: Result of Q7.

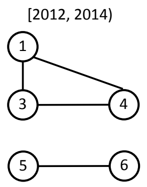


Figure 9: Result of Q8.

then temporally aggregates the resulting TGraph by 2 years. Finally, pagerank() is computed for each vertex in the result.

```
Q7:  TSelect Any V [vid, pagerank() as pr] ;
             Any E [vid1, vid2]
     From    T1 TAnd T2
     TWhere  Start >= 2012 And End < 2014
     TGroup by 2 years
```

Observe that TAnd and TGroup use the same specification in the TSelect clause, that is, we cannot decouple structural aggregation behavior of the two operations in a query of this kind.

Consider next query $Q8$ that is similar to $Q7$, but uses nesting to decouple structural aggregation of TSelect from that of TGroup. The result of executing $Q8$ is shown in Figure 9.

```
Q8:  TSelect    All V [vid, pagerank() as pr];
                All E [vid1, vid2]
     From       ( TSelect Any V [vid] ;
                          Any E [vid1, vid2]
                  From    T1 TAnd T2
                  TWhere  Start >= 2012
                  And     End < 2014 )
     TGroup     by 2 years
```

Trend analytics. We argued in the introduction that it is important to support analysis of trends in evolving graphs. The Portal query language makes this sophisticated analysis possible. Query $Q9$ illustrates this; it invokes the snapshot analytic pagerank() on T1, and then computes the trend in these values across all snapshots.

```
Q9:  TSelect Any V[vid, trend(pr) as tr, max(pr) as mx];
             Any E[vid1, vid2]
     From    ( TSelect V[vid, pagerank() as pr];
                       E[vid1, vid2]
               From    T1 )
     TGroup   by Size
```

Consider the use of the *trend analytic* function trend(pr), which aggregates the sequence of PageRank scores of each vertex. In our implementation we use a common definition of trend: compute the slope of the least squares line using linear regression, making an adjustment when a vertex value is missing. While this is the only trend analytic we currently support, we are working on an API that will allow developers to implement custom trend analytics, taking attributes of both atomic and complex type as input, and computing a value of either an atomic or a complex type.

We invoke trend(pr) alongside max(pr) in $Q9$, to highlight the difference between a trend analytic and value aggregation. Syntactically, the two look the same, but the difference is that max(pr) does not account for the temporal order of values being aggregated, while trend(pr) does. Based on this distinction, it is not meaningful to invoke a trend analytic when structural aggregation is due solely to temporal join (TAnd) or (TOr), but only as part of a query that does temporal aggregation (TGroup).

Finally, recall from our discussion of temporal aggregation that TGroup by Size produces a 1-snapshot TGraph. It is convenient, although not required, to use this feature in a query such as $Q9$, since a temporal trend may be computed over a window of any size.

3.4 Loading data and inspecting results

Loading filesystem data. Views. Queries $Q1$ through $Q9$ refer to TGraph variables in the From clause. The value of a TGraph variable is assigned by the define view statement. This value may be loaded from the file system, as in query $Q10$, or it may correspond to a view, as in query $Q11$.

```
Q10:  Create Tview T1 as
        (TSelect V[vid:int, name:str, salary:int];
                 E[vid1:int, vid2:int, cnt:int]
         From    path/to/directory)

Q11:  Create Tview T4 as (TSelect V; E
         From    T1
         TWhere  Start >=  2010)
```

Note that when data is loaded into a TGraph variable from the file system, as in $Q10$, the TSelect clause must specify the structural schema. When a TGraph variable takes on a value computed by a query, its structural schema is determined by the query result. In $Q11$, the structural schema of the view T4 is the same as that of T1.

Inspecting results with SQL. Suppose that the result of $Q9$ is assigned to T5, with the structural schema V($\underline{\text{vid}}$:int, tr:float, mx:float) ; E($\underline{\text{vid1}}$:int, $\underline{\text{vid2}}$:int). The SQL query below shows vid and tr values of 20 vertices with the most significantly increasing pagerank trend.

```
Q12:  Select    VF.vid, VF.tr
      From      T5.toVerticesFlat() as VF
      Order by  tr
      Limit     20
```

The important part of Q12 is the use of T5.toVerticesFlat() in the From clause. This is a multi-step operation provided by the Portal framework, which starts by collecting all vertices in the union of snapshots of T5 into a single nested vertex collection, and associating $\underline{\text{vid}}$ with a time-indexed map of vertex attributes. Next, the nested collection is flattened into VF ($\underline{\text{vid}}$:int, $\underline{\text{start}}$:date, $\underline{\text{end}}$:date, tr:float, mx:float). VF can be used in SQL queries. The Portal framework also provides an operation that returns a flattened collection of edges, called toEdgesFlat().

4. SYSTEM

Our Portal system implementation builds on GraphX, an Apache Spark library, as depicted in Figure 10. Green boxes indicate built-in components, while blue are those we added for Portal. We selected Apache Spark because it is a popular open-source system, and because of its in-memory processing approach. All language operators on TGraphs are available through the public API of the Portal library, and may be used like any other library in an Apache Spark application.

Portal query execution follows the traditional query processing steps: parsing, logical plan generation and verification, and physical plan generation. Portal re-uses and extends SparkSQL abstractions for these steps. A Portal

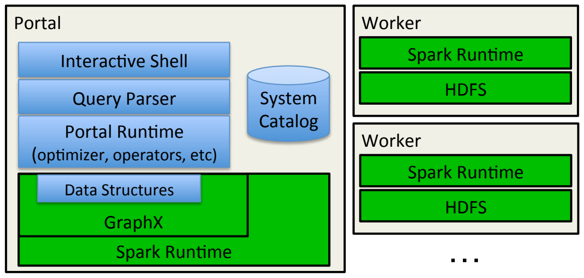


Figure 10: Portal system architecture.

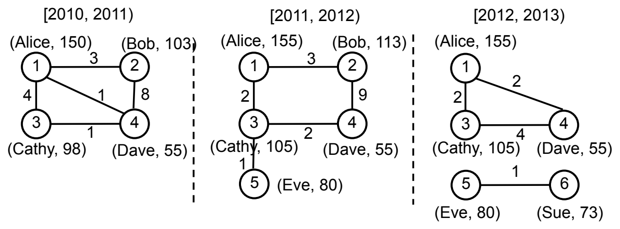


Figure 11: SG representation of T1 from Figure 1 (partial).

query is rewritten into a sequence of operators, and some operators are reordered to improve performance (Section 4.1). We developed several different physical representations (Section 4.2) and partitioning strategies (Section 4.3) that are selected at the physical plan generation stage. The TGraphs are read from the distributed file system HDFS and processed by Spark Workers, with the tasks assigned and managed by the runtime. The System Catalog contains information about each data set, including the temporal and structural schema, and temporal bounds.

4.1 Operator Reordering

The logical order of evaluation of a Portal query without nesting is as follows: (1) temporal selection (TWhere); (2) temporal join (TAnd or TOr); (3) temporal aggregation (TGroup); (4) projection, computation of attribute value aggregated and analytics (TSelect). While the logical order of operations is predetermined, some operators can be reordered without affecting the result, but with potential differences in performance, while others cannot.

TGroup before join. Temporal aggregation produces a smaller number of snapshots than its input. Since the performance of joins depends on the number of snapshots being structurally aggregated, a smaller number of snapshots will lead to better performance. Consider this query:

```
TSelect Any V; Any E
From    T1 TOr T2
TGroup  by 5 years
```

Assuming that T1 and T2 have 1-year resolution, TGroup will reduce the number of graphs to be aggregated by a factor of 5. Therefore, it seems beneficial to apply TGroup to each T1 and T2 first, and to then compute TOr of the result. Unfortunately this reordering is not always possible. For T1 and T2 in Figures 1 and 2, TGroup produces TGraphs with temporal schema ([2010, 2015), [2015, 2020)) for T1 and ([2008, 2013), [2013, 2018)) for T2. The two TGraphs are not temporally union-compatible and so cannot be joined.

TAnd before temporal selection. Temporal intersection between two graphs with a small temporal overlap can cut down on graph loading time significantly. Consider this query:

```
TSelect All V; All E
From    T1 TAnd T2
```

If temporal bounds of T1 and T2 are known, the query can be rewritten with an explicit temporal selection matching the length of the overlap. For T1 and T2 in Figures 1 and 2, the rewriting is:

```
TSelect All V; All E
From   T1 TAnd T2
TWhere T1.Start >= 2010 And T1.End < 2014
       And T2.Start >= 2010 And T2.End < 2014
```

This rewriting reduces the loading size of T1 and T2 by $\frac{1}{3}$ (ignoring temporal skew). As we will show in Section 5, loading time is a significant portion of the over-all time, and reducing it will have a noticeable effect on performances.

Projection before aggregation or join. Recollect that temporal aggregation and join operate not only on graph structure, but also on vertex and edge attributes. In some cases, the values of some or all of those attributes are irrelevant to the query, and attribute aggregation can be greatly reduced or eliminated. Consider query Q6 from Section 3, reproduced here for convenience:

```
TSelect   All V[vid, pagerank() as pr];
          Any E[vid1, vid2, sum(cnt)]
From      T1 TOr T2
```

Vertices of T1 and T2 have non-key attributes name and salary, which are not used in this query. Therefore, to improve performance, we can project the vertices of T1 and T2 to V [vid] prior to evaluation of the TOr operation.

4.2 Data Representation

We developed several in-memory representations of evolving graphs to explore the trade-offs of compactness, parallelism, and support of different query operators.

SnapshotGraph (SG). The simplest way to represent an evolving graph is by representing each snapshot individually, a direct translation of our logical data model. We call this data structure SnapshotGraph, or SG for short. An example of an SG is depicted in Figure 11. SG is a collection of snapshots, where vertices and edges store the attribute values for the specific time interval. A TSelect operation on this representation is a slice of the snapshot sequence, while TGroup and temporal joins (TAnd and TOr) require a group by key within each aggregate set of vertices and edges.

While the SG 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 [22]. In a distributed architecture, however, this data structure provides some benefits as operations on it can be easily parallelized, by assigning different snapshots to different workers, or by partitioning a snapshot across workers.

MultiGraph (MG). To take advantage of high similarity between snapshots, we developed another data structure called MultiGraph, or MG for short (Figure 12). MG stores the evolving graph as a single graph, with one vertex for all

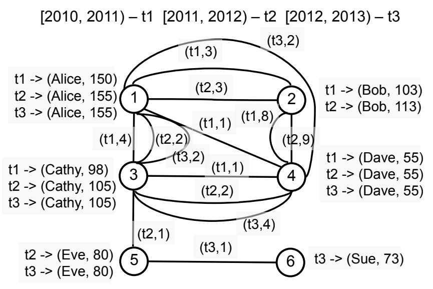


Figure 12: MG for T1 (partial).

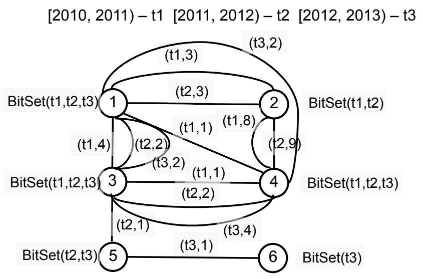


Figure 13: MGC for T1 (partial).

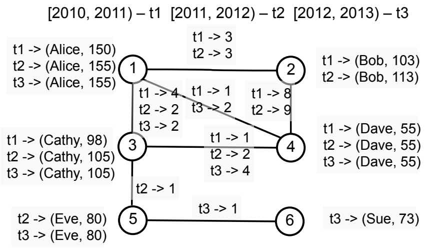


Figure 14: OG for T1 (partial).

time periods, but with one edge per period where it exists. Because our goal is to represent both topological and attribute information, we need to store not only vertex presence or absence (which can be easily accomplished by an existence string, like in [14], or by bit sets), but also the values of vertex attributes at each time period. MG vertex attribute, thus, is a map of time indices that represent intervals to corresponding values. Edge attributes are tuples of the time index and the corresponding value. Vertices typically change infrequently, so storing each vertex only once reduces the total number of vertices by about 80% in our data. Some of these savings, however, are taken up by the storage of a more complex map data structure for attribute values, compared to a single attribute as in SG.

The implementation of some of the Portal operations in MG is more complex than in SG. TSelect is a subgraph operation that operates on all vertices and edges. TGroup on vertices is a combination of transform and filter operations, since the vertices are already aggregated across the whole time period, but the edges, like in SG, are grouped by key within their respective sets. Implementation of snapshot analytics like PageRank is done in batch mode, similar to [22], by computing the values and sending messages between vertices for all time periods at once.

Note that for a large subset of the queries, attribute information is not used, and only the topology is important. Thus, we can store vertex attributes in a separate collection (column store), removing the attribute map and replacing it with existence bit sets instead. This is the essence of the special case of MultiGraph, called **MultiGraphColumn (MGC)**, depicted in Figure 13. The MGC representation allows storage of an arbitrary number of vertex attributes without using complex per-vertex lists, read from disk only as needed, and so amenable to lazy evaluation, an important performance optimization in Apache Spark. Further compression can be achieved by storing vertex attribute values only once across all time periods in which they are the same, similar to how temporal databases represent this type of data (e.g., see [24]). The drawback of this approach is that decompression is required to support, for example, the TGroup operation.

OneGraph (OG). The most topologically compact representation is to store each vertex *and* each edge only once for the whole evolving graph, by taking a union of the snapshot vertex and edge sets. The OneGraph data structure, or OG for short, uses this representation in our system. Similar to MG, vertex and edge attributes are stored in time-indexed maps (Figure 14). Compared to MG, this leads to storing about 75% fewer edges. The OG data structure provides some benefits in addition to compactness, since it reduces the total communication between vertices in Pregel-based analytics in batch mode. The drawback is that OG is much denser than individual snapshots. As with MG, TSelect is a subgraph operation, and TGroup is a transform and filter operation — for both vertices and edges.

Similar to MGC, **OneGraphColumn (OGC)** uses a single graph to represent the union of vertices and edges, with bit sets for presence information, while attribute information is stored separately. This is not as compact as storing attributes within the graph elements, but is faster in many operations where only graph topology is required.

4.3 Partitioning Strategies

Graph partitioning can have a tremendous impact on system performance. A good partitioning strategy needs to (1) be balanced, assigning an approximately equal number of units to each partition, and (2) limit the number of cuts across partitions, to reduce cross-partition communication.

There are two basic types of graph partitioning strategies. Vertex-cut (also known as edge partitioning) distributes edges across the available machines and replicates vertices as necessary, while edge-cut (vertex partitioning) does the opposite. Vertex-cut approaches have been shown to have better performance [11], and are the strategies of choice in Graph X. In Portal, we support six different vertex-cut strategies, which are applied prior to the operation but after loading, and can be re-applied at any point.

Canonical Random Vertex Cut (CRVC). The source and destination ids of a vertex are hashed in a canonical direction and the result is distributed among the available partitions. The result is a random vertex cut that co-locates all edges connecting a given pair of vertices, regardless of direction. This strategy is available in GraphX and was used without modification.

2D Edge (E2D). A sparse edge adjacency matrix is partitioned in two dimensions (Figure 15), guaranteeing a $2\sqrt{n}$ bound on vertex replication, where n is the number of partitions. E2D can provide good performance for Pregel-style analytics. This strategy is available in GraphX and was used without modification.

Naive Temporal. Let n be the number of snapshots and p be the number of partitions. If $n \geq p$, then snapshot i is placed into partition $i\%p$. Otherwise, multiple partitions are grouped into a *run*, and the CRVC strategy is used to partition a snapshot within each run.

Consecutive Temporal. Let n be the number of snapshots and p be the number of partitions. If $n \geq p$, then temporally consecutive snapshots are assigned to the same

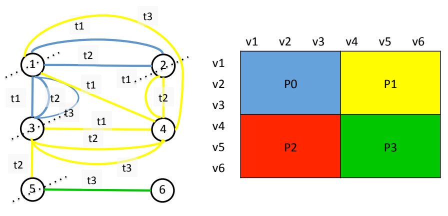


Figure 15: E2D with 4 partitions.

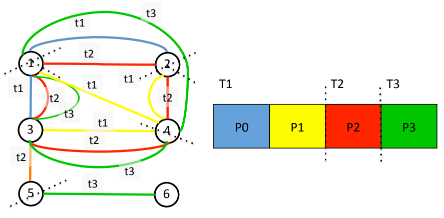


Figure 16: Consecutive with 4 partitions, 3 snapshots.

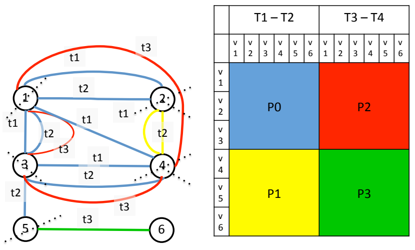


Figure 17: E2D-Temporal with 4 partitions, 4 snapshots, 2 runs.

partition. Otherwise, each snapshot is assigned an equal number of consecutive partitions, and CRVC is used to partition the snapshot (Figure 16).

Many networks exhibit strong temporal skew, with later snapshots being significantly larger than earlier ones. Consecutive temporal strategy may result in an unbalanced partitioning for networks with large skew.

Furthermore, the number of cuts for a vertex is the number of intervals in which it exists with degree > 0. In the worst case, a vertex will be cut n times. Therefore, if vertices persist across many time intervals, this strategy is not a good choice.

Hybrid strategies. Hybrid strategies combine elements of temporal and structural criteria. We define two such strategies, E2D-Temporal (Figure 17) and CRVC-Temporal. With both strategies, we create runs of temporally consecutive snapshots, assign an equal number of partitions to each run, and then use E2D or CRVC within each run.

These strategies are specifically designed for time-based operations such as aggregation to co-locate edges that are to be grouped. Run width is determined based on the operation. For TGroup, we take the number of snapshots in a group as the run width. For other operations the default run width is 2.

Note that not all partitioning strategies can be applied to every data structure. For example, only purely structural strategies can be applied to OG. We report on the experimental effectiveness of the strategies in Section 5.

5. EXPERIMENTAL EVALUATION

Experimental environment. All experiments in this section were conducted on an 8-slave in-house Open Stack cloud, using Linux Ubuntu 14.04 and Spark v1.4. 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. In cases where the goal was to evaluate a specific operation in isolation, we used warm start, which consisted of materializing the graph upon load. Each experiment was conducted 3 times, 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.

Data. We evaluate performance of our framework on two real open-source datasets. DBLP [1] contains co-authorship information from 1936 through 2015, with over 1.5 million author nodes and over 6 million undirected co-authorship edges. Total data size: 250 MB. nGrams [13] contains word co-occurrence information from 1520 through 2008, with over 1.5 million word nodes and over 65 million undirected co-occurrence edges. Total data size: 40 GB.

The nGrams dataset is of comparable size to the LiveJournal dataset in [30] and is commensurate with our cluster size. DBLP and nGrams differ not only in size, but also in the evolutionary properties: co-authorship network nodes and edges have limited lifespan, while the nGrams network grows over time, with nodes and edges persisting for long duration. All figures in the body of this section are on the larger nGrams dataset. Refer to the Appendix for the DBLP figures, which show similar trends as nGrams.

5.1 Number of partitions

In Spark applications, one of the most influential performance drivers is the choice of the number of partitions. The default number of partitions, based on the Hadoop block size, proved inefficient in practice. We extended GraphX for parallel reading of multiple files with a custom number of partitions. Further, we conducted a tuning experiment, and developed a dynamic data size-based estimator of the number of partitions.

The tuning experiment consisted of running a simple TSelect query with each data structure, varying the number of partitions on load for a range of data sizes. In each data structure, we observe the same trend – as the number of partitions is increased for a given data size, system performance quickly improves, but then starts to deteriorate (Figure 18). There is roughly a linear relationship between data size and the best number of partitions (see Figure 26 in the Appendix), which allowed us to fit a linear function and use it to tune the number of partitions in all experiments.

5.2 Data loading

To understand how different data structures perform as a function of data size, we ran the following query on nGrams:

```
TSelect V[vid:int, word:str];
        E[vid1:int, vid2:int, cnt:int]
From    data/nGrams
Into    nGrams
TWhere  Start >= x and End < y
```

where x and y parameters were varied to achieve approximate desired data sizes for edge files. Our file format is a node file and an edge file per snapshot, with a node/edge per line, respectively. This format favors the SG data structure because no data transformation, such as aggregation, is required. This is substantiated experimentally, as can be seen in Figure 19. SG is the fastest data structure for data

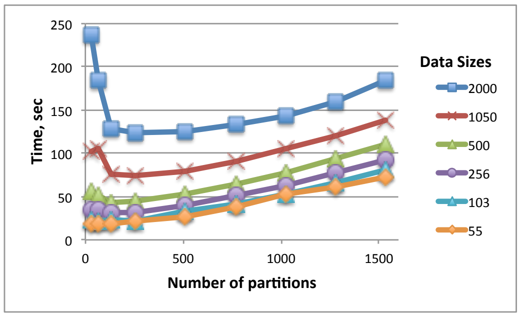


Figure 18: Load time vs. # of partitions.

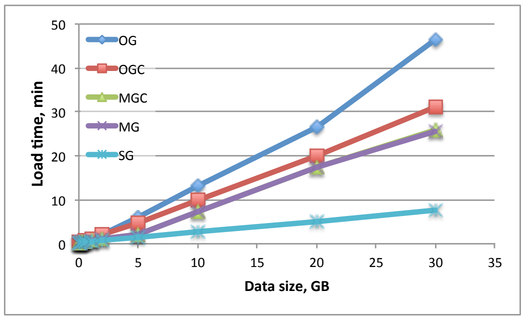


Figure 19: Load time vs. data size, nGrams.

loading, although all exhibit a linear increase as a function of data size.Miao et al. [22] showed that data format plays a significant role in load performance. This is supported by our results in Figure 19, and warrants development of alternative file formats, which is in our immediate plans.

As we will see, load/materialize time makes up a significant portion of the over-all time. For this reason, all experiments that evaluate performance of individual Portal operators are executed with a warm start.

5.3 TSelect with TGroup

To investigate the comparative performance of different data structures and partition strategies on the TGroup operation, we used the following query:

```
TSelect Any V[vid, any(word)];
        Any E[vid1, vid2, sum(cnt) as score]
From    nGrams
TWhere  Start >= x And End < y
TGroup  by 8 years
```

We kept the aggregation time window fixed (8 years) and varied the total number of snapshots in powers of 2. Each snapshot contains about 13 million edges.

Recollect that TGroup requires a union and group by operation in SG, a combination of transform with filter and group by for MG, and transform with filter for OG. Due to compactness, MGC outperforms MG and OGC outperforms OG. All data structures and partitioning strategies show linear increase in structural aggregation time as the number of snapshots grows. As expected, because OGC is an already aggregated data structure, it outperforms MGC and SG for the TGroup operation by up to two orders of magnitude. This is shown in Figure 20, where we present only the best-performing variant (column-store) and partitioning strategy for each data structure.

To compare between partitioning strategies, consider Figure 21, which contains performance for each data structure and strategy at 64 snapshot aggregation. (The same pattern is observed at all aggregation sizes.) Partitioning strategies improve performance for MGC on this operation — all outperform the default no partitioning case by 20% or more. This can be explained by the group by operation that is performed on the edges. E2D always places two edges with the same key in the same partition, which leads to least cross-partition communication during aggregation. Hybrid strategies use the aggregation window as the width of the run, also placing the edges that are to be grouped in the same set of partitions. The differences between E2D and the hybrid strategies for MGC are not significant. E2D also does best for SG, but the gains are smaller than for MGC.

We observed the same trend for the TGroup query with All semantics — see Appendix.

To demonstrate why we use warm start, consider Figure 22 where the fastest partition condition for each data structure is depicted (and is None in all 3 cases), and note that loading time dominates the overall time, which also includes partitioning, temporal aggregation, and materialization. Based on this figure alone, one would conclude that SG is the most efficient data structure. However, recall that the file format favors SG, and so no conclusions about the performance of data structures for individual operations should be drawn from cold start total time alone.

OGC outperforms the other data structures when we fix the data size and vary the aggregation width. OGC is insensitive to aggregation width, with a slight improvement as the width increases. SG and MGC show a linear increase, with a drop for the final aggregation by the whole size. This trend is observed in both datasets, see Appendix for figures.

5.4 TAnd with All

TAnd is a binary operation, and thus co-partitioning of the TGraphs should improve performance. In this experiment we use the query:

```
TSelect All V[vid, max(word)];
        All E[vid1, vid2, max(cnt) as score]
From    ( TSelect V; E
          From nGrams
          TWhere  Start >= x And End < y )
        TAnd
        ( TSelect V; E
          From nGrams
          TWhere  Start >= n And End < m )
```

We are selecting the two graphs from the same dataset and varying the amount of temporal overlap (n - m). Note that this is the worst-case scenario for structural aggregation with All, since the size of the intersection among the corresponding pairs of snapshots is the highest possible. Note that, while this query is a good candidate for query rewriting, we did not optimize it in this experiment, but executed it directly as specified.

TAnd performs structural aggregation of for each snapshot pair in SG, and one structural aggregation in all other data structures. As the temporal overlap between the two graphs increases, we expect OGC to outperform SG, and this is what we observe experimentally (Figure 23). SG outperforms other representations when overlap is 7% or lower, while OGC significantly outperforms SG and MGC for higher overlap values.

Partitioning has a mild benefit for MGC, especially with

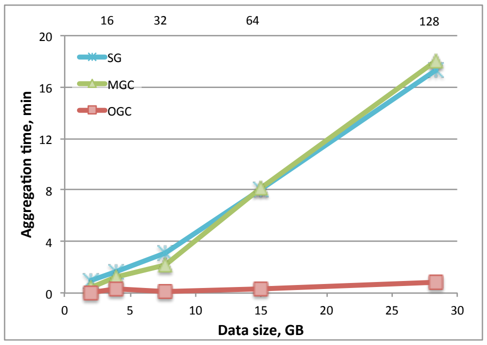


Figure 20: TGroup with Any (warm start).

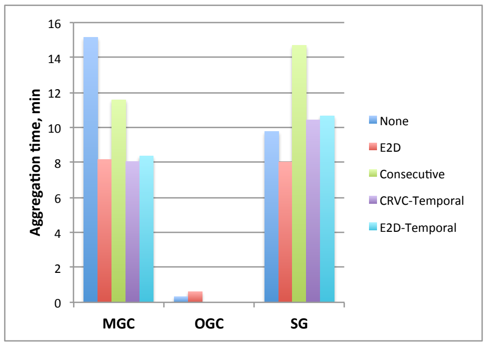


Figure 21: TGroup by partitioning strategy.

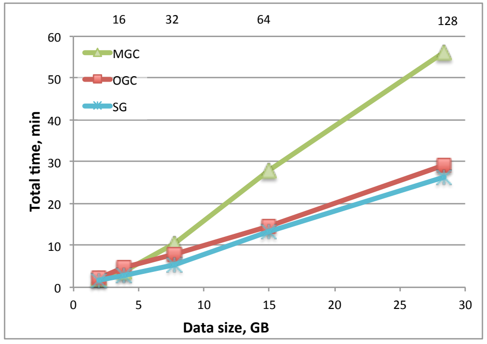


Figure 22: TGroup with Any (cold start).

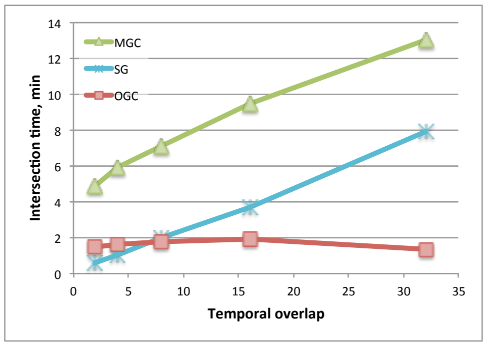


Figure 23: TAnd time vs. temporal overlap.

the E2D strategy, as it does for OGC and SG. See Figure 30 in the Appendix for side-by-side comparison.

TOr performance with Any exactly matches that of TAnd with All above, and we do not show it here.

5.5 PageRank

Snapshot analytics like PageRank are implemented using the Pregel API in GraphX, with batch mode for MGC and OGC. We used the following query to evaluate data structure performance over varying number of snapshots:

```
TSelect V[vid, pagerank()];
        E[vid1, vid2]
From    nGrams
TWhere  Start >= x And End < y
```

PageRank was executed for 10 iterations or until convergence, whichever came first. Performance of Pregel-based algorithms depends heavily on the partition strategy, with best results achieved where cross-partition communication is small. For this reason, we evaluated only no partitioning and E2D.

SG performs better than the other data structures in this experiment, contrary to our expectation that batch mode of MGC and OGC would be faster (Figure 24). This can be explained by MGC and OGC using significantly more cross-partition communication due to the following factors:

1. Each individual snapshot is less dense than the aggregate (although this depends on the rate of change), and dense graphs do worse with Pregel analytics.
2. Individual snapshots are smaller and take fewer partitions, so less communication happens across partitions.
3. PageRank gets faster as vertex values converge, because those vertices stop sending out new messages. In OGC/MGC a vertex converges only when it does so in all snapshots.

E2D partitioning leads to performance improvements in all data structures except MGC.

5.6 TSelect with trend(pagerank())

All the experiments so far evaluated performance of individual Portal operations. We conclude this section with a cold-start execution of the query:

```
Select vid, pr
From (TSelect Any V[vid, trend(prank) as pr];
              Any E
      From (TSelect All V[vid, pagerank() as prank];
                    All E
            From nGrams
            TWhere Start >= x And End < y
            TGroup by 8 years)
      TGroup by size).toVerticesFlat()
Order by pr
Limit 10
```

As we saw above, SG outperforms other representations for data load and for PageRank, while OGC is very efficient for temporal aggregation. This query combines all of these operations, and adds a trend analytic, and a transformation of the vertices of the result into a flat vertex relation.

SG with no partitioning, and OGC with E2D show comparable performances, as seen in Figure 25.

In summary, no one data structure is most efficient across all operations. SG is most efficient for data load, because our file format favors this data structure, and for PageRank. OGC is most efficient for temporal group and join. The two data structures perform comparably for the complex query. E2D is the most efficient partitioning method in most cases, and E2D-Temporal is a close second.

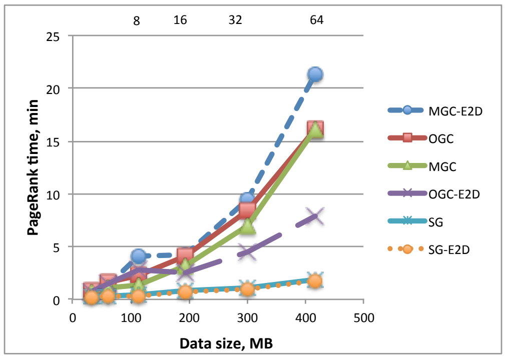


Figure 24: PageRank time.

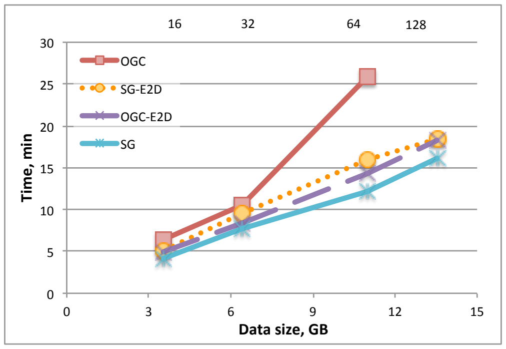


Figure 25: TGroup, PageRank, trend.

6. RELATED WORK

Querying and analytics. Portal is the first proposed declarative query language for evolving graphs. There has been much recent work on analytics for evolving graphs, see [2] 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. [14] propose a query model for discovering subgraphs that match a specific spatio-temporal pattern. Chan et al. [6] query evolving graphs for patterns represented by waveforms. Semertzidis et al. [28] focus on historical reachability queries.

Our work shares motivation with Miao et al. [22], who developed an in-memory execution engine for temporal graph analytics called ImmortalGraph. Unlike Miao et al., who focus on in-memory layout and locality-aware scheduling mechanisms, we work in a distributed processing environment. A further difference is that our work is in scope of Apache Spark, a widely-used open source platform, while ImmortalGraph is a proprietary stand-alone prototype.

Our work differs from query processing on dynamic graphs (e.g., [23]), where the history of changes is not important, and from mining evolving graph streams (e.g., [20]), where the focus is on discovering significant changes.

Data representation. The basic building block in Portal is a snapshot, which naturally limits the resolution at which changes can be retrieved. This deliberate choice is in contrast with delta-based approaches [15, 16, 22].

Khurana and Deshpande [15] investigate efficient physical representations using deltas to support snapshot retrieval. Their in-memory GraphPool maintains a single representation of all snapshots, and is thus similar to our MultiGraph and OneGraph. GraphPool goes further and stores only dependencies from a materialized snapshot when deltas are small. We do not take this step because evaluation of queries involving multiple snapshots, such as TGroup, requires fully materialized views in memory. The snapshot group method of [22] is similar to DeltaGraph in [15].

Ren et al. [26] develop an in-memory representation of evolving graphs based on representative graphs for sets of snapshots. Representative graphs can be computed using structural aggregation in Portal, and so this work provides additional motivation for our temporal aggregation and join operators. Our OneGraph can be thought of as a representative graph for the whole selected time period. In the future we may consider a hybrid approach similar to [26], materializing multiple OneGraphs for different sets of snapshots if a single representation becomes too dense.

Semertzidis et al. [28] develop a version graph, where each node and edge are annotated with the set of time intervals in which they exist. This is similar to our OneGraph, but we also store non-topological attributes. We also do not compress consecutive existence intervals with the same value into one, because this would make TGroup and snapshot analytics more complex and more difficult to parallelize.

Boldi et al. [5] present a space-efficient non-delta approach for storing a large evolving Web graph that they harvested. Their approach for encoding the presence or absence of nodes and edges at each time interval using bits is similar to our MultiGraph and its use of BitSets. The primary difference is that their work represents purely topological information and does not address vertex and edge attributes.

Distributed frameworks. We build upon GraphX [12], which provides an API for working with regular graphs (snapshots) in Apache Spark, but without the time dimension. To improve performance, we modified the GraphX graph loading code, enabling concurrent distributed multi-file loading with a tuned number of partitions. We use built-in GraphX partitioning and develop additional vertex-cut strategies that incorporate the temporal dimension. Another relevant framework is SparkSQL [31], which provides mid-query optimization of distributed execution. Mid-query optimization is in our immediate plans for Portal. In SparkSQL tables to be joined can be co-partitioned at creation time. This is similar to what our data structures and partition strategies achieve for TGroup and temporal join.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented Portal, a declarative query language for evolving graphs. We also proposed an implementation of Portal in scope of Apache Spark, a distributed open-source processing framework. We implemented several physical representations of evolving graphs, and several partitioning strategies, and studied their relative performance for Portal operators.

Our experiments demonstrate interesting trade-offs between spatial and temporal locality. This work opens many avenues for future work. It is in our immediate plans to start work on a query optimizer for Portal. We will also implement and evaluate additional TGraph representations that explore the trade-off between density and compactness, and between temporal and structural locality. Finally, we are working on extending the class of trend analytics, and on optimizing the performance of snapshot and trend analytics.

8. REFERENCES

[1] DBLP, computer science bibliography. `http://dblp.uni-trier.de/`. [Online; accessed 14-November-2015].

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

[3] S. Asur, S. Parthasarathy, and D. Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. *TKDD*, 3(4), 2009.

[4] A. Beyer, P. Thomason, X. Li, J. Scott, and J. Fisher. Mechanistic insights into metabolic disturbance during type-2 diabetes and obesity using qualitative networks. *T. Comp. Sys. Biology*, 12:146–162, 2010.

[5] P. Boldi, M. Santini, and S. Vigna. A Large Time-Aware Web Graph. *ACM SIGIR Forum*, 42(2):33–38, 2008.

[6] J. Chan, J. Bailey, and C. Leckie. Discovering correlated spatio-temporal changes in evolving graphs. *Knowledge and Information Systems*, 16(1):53–96, 2008.

[7] J. Chan, J. Bailey, and C. Leckie. Discovering correlated spatio-temporal changes in evolving graphs. *Knowl. Inf. Syst.*, 16(1):53–96, 2008.

[8] T. A. S. Foundation. Apache giraph. `http://giraph.apache.org/`. [Online; accessed 14-November-2015].

[9] M. Goetz, J. Leskovec, M. McGlohon, and C. Faloutsos. Modeling blog dynamics. In *ICWSM*, 2009.

[10] J. Gonzalez, Y. Low, and H. Gu. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[11] J. Gonzalez, Y. Low, and H. Gu. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 17–30, 2012.

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

[13] Google. The google books ngram dataset. `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`. [Online; accessed 14-November-2015].

[14] A. Kan, J. Chan, J. Bailey, and C. Leckie. A Query Based Approach for Mining Evolving Graphs. In *Eighth Australasian Data Mining Conference (AusDM 2009)*, volume 101, Melbourne, Australia, 2009.

[15] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In *ICDE*, pages 997 – 1008, Brisbane, QLD, 2013.

[16] G. Koloniari. On Graph Deltas for Historical Queries. In *Proceedings of 1st Workshop on Online Social Systems (WOSS) 2012*, Istanbul, Turkey, 2012.

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

[18] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *TWEB*, 1(1), 2007.

[19] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins. Microscopic evolution of social networks. In *ACM SIGKDD*, pages 462–470, 2008.

[20] Z. Liu and J. Yu. Discovering burst areas in fast evolving graphs. In H. Kitagawa, Y. Ishikawa, Q. Li, and C. Watanabe, editors, *Database Systems for Advanced Applications*, volume 5981 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin Heidelberg, 2010.

[21] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, pages 135–146, 2010.

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

[23] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *ACM SIGMOD*, page 145, 2012.

[24] H. Muller, P. Buneman, and I. Koltsidas. XArch : Archiving Scientific and Reference Data. In *ACM SIGMOD*, pages 1295–1298, New York, NY, 2008.

[25] P. Papadimitriou, A. Dasdan, and H. Garcia-Molina. Web graph similarity for anomaly detection. *J. Internet Services and Applications*, 1(1):19–30, 2010.

[26] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. *PVLDB*, 4(11):726–737, 2011.

[27] P. Sarkar, D. Chakrabarti, and M. I. Jordan. Nonparametric link prediction in dynamic networks. In *ICML*, 2012.

[28] K. Semertzidis, K. Lillis, and E. Pitoura. TimeReach: Historical Reachability Queries on Evolving Graphs. In *EDBT*, pages 121–132, Brussels, Belgium, 2015.

[29] J. M. Stuart, E. Segal, D. Koller, and S. K. Kim. A gene-coexpression network for global discovery of conserved genetic modules. *Science*, 5643(302):249—255, 2003.

[30] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, and U. C. Berkeley. GraphX : A Resilient Distributed Graph System on Spark. In *GRADES*, New York, New York, USA, 2013.

[31] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, I. Stoica, and U. C. Berkeley. Shark : SQL and Rich Analytics at Scale. In *ACM SIGMOD*, 2013.

[32] L. Yang, L. Qi, Y. Zhao, B. Gao, and T. Liu. Link analysis using time series of web graphs. In *CIKM*, pages 1011–1014, 2007.

APPENDIX

Plots and discussion in this section complement experimental results presented in Section 5.

Figure 26 complements Figure 18, in which we presented the impact of number of partitions on the overall performance. In Figure 26 we show the linear trend between the data size and the number of partitions that leads to fastest execution. The linear equation this fit produces is used in the partition estimator during graph loading, except for very small data sizes.

In Figure 27 we show that TGroup with All semantics exhibits the same behavior in all data structures as TGroup with Any semantics in Figure 20. This is to be expected as both All and Any use the same group by operation on the data, but with different restrictions after grouping.

Another aspect of aggregation is the impact of an aggrega-

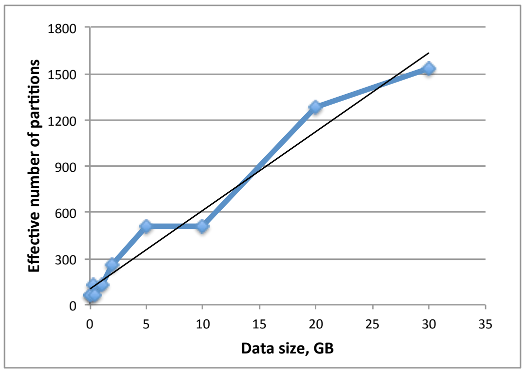


Figure 26: Effective number of partitions.

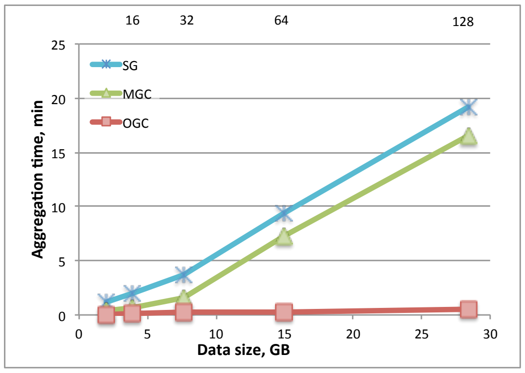


Figure 27: TGroup with All (warm start).

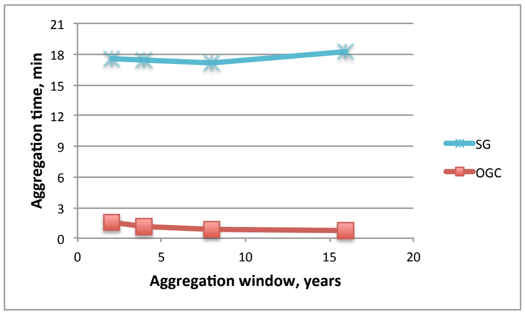


Figure 28: TGroup by width, nGrams.

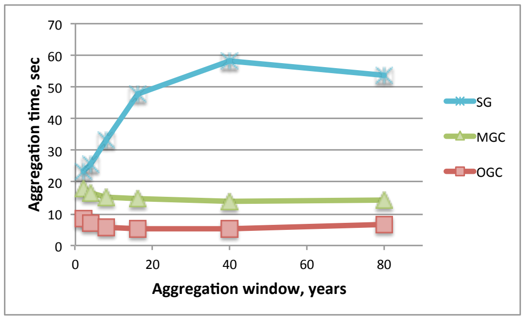


Figure 29: TGroup by width, DBLP.

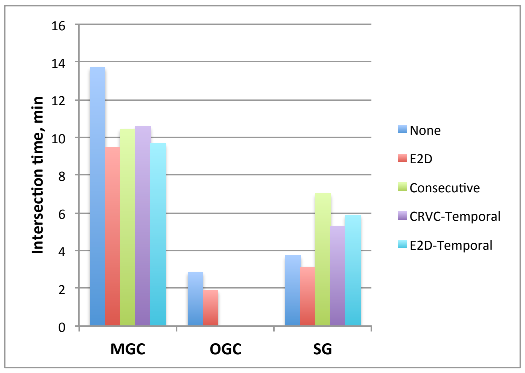


Figure 30: TAnd by partition strategy, nGrams.

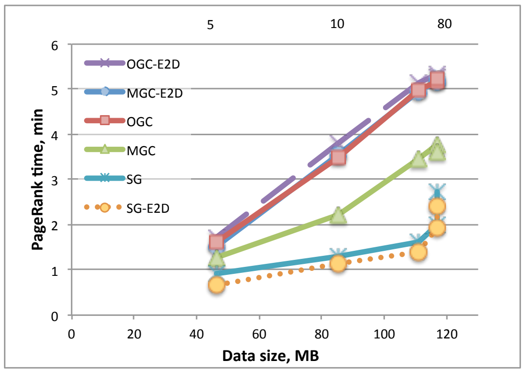


Figure 31: PageRank time, DBLP.

tion width. Consider Figures 28 and 29 for the nGrams and the DBLP datasets, respectively. We fix the graph interval (128 and 80 snapshots, respectively) and vary the width in powers of 2 up to total size. For each data structure we picked the most efficient partition strategy based on the experiment in Section 5.3, Figure 21. Observe that OGC and MGC are nearly insensitive to the aggregation width, with small gains as the width increases. In contrast, SG performance logarithmically worsens as the width increases in the DBLP dataset, but is largely unchanged for the width values we evaluated in the nGrams dataset.

Figure 30 complements Figure 23. We show that, similar to aggregation, partitioning improves performance for temporal joins. This finding is expected because co-partitioning of two graphs along the same criteria improves the structural grouping operation that is carried out. E2D partition strategy provides the best performance, but the difference between it and the hybrid strategies is not significant.

Finally, Figure 31 shows PageRank performance on the DBLP dataset. Due to the very pronounced skew in the DBLP dataset, the number of snapshots was drawn from the most recent to the left on the timeline. The same number of partitions was used for MGC/OGC because the data size did not differ significantly between different time intervals. The general trend observed is the same as in the nGrams dataset, with the only difference that at these small sizes the E2D strategy does not produce an improvement in performance for MGC and OGC.