Towards a distributed infrastructure for evolving graph analytics

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 present our ongoing work on the Portal system, an open-source distributed framework for evolving graphs. Portal streamlines exploratory analysis of evolving graphs, making it efficient and usable, and providing critical tools to computational and data scientists. Our system implements a declarative query language by the same name, which we briefly describe in this paper. Our basic abstraction is a TGraph, which logically represents a series of adjacent snapshots. We present different physical representations of TGraphs and show results of a preliminary experimental evaluation of these physical representations for an important class of evolving graph analytics.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

evolving graphs, graph analytics, distributed computation

1. INTRODUCTION

The development of SQL, a declarative query language for relational data analysis, had a tremendous impact on the usability of database technology, leading to its wide-spread adoption. At the same time, the separation between the logical and the physical representations paved the way for powerful performance optimizations. The motivation for our research is to provide a similar tool for analyzing *evolving graphs*, an area of interest in many research communities, including the Web and social networks, sociology, epidemiology, and others.

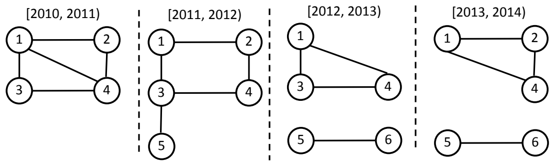


Figure 1: TGraph T1 with 4 snapshots.

Analysis of evolving graphs has been receiving increasing attention, with most progress taking place in the last decade [1, 6, 10, 17, 19, 21]. Some areas where evolving graphs are being studied are social network analysis [8, 14, 15, 20], biological networks [2, 3, 22] and the Web [7, 18].

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.

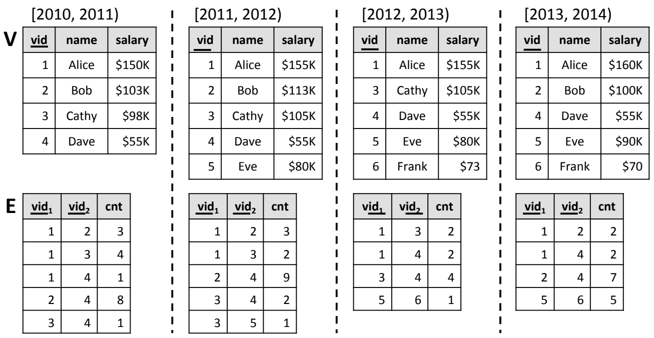
In this paper we present our ongoing work on the Portal system, an open-source distributed framework built on top of Apache Spark, that fills this gap. Portal streamlines exploratory analysis of evolving graphs, making it efficient and usable, and providing critical tools to computational and data scientists. Importantly, Portal implements a declarative query language for evolving graphs. We briefly describe our query language (Section 2) and the system (Section 3) that implements it. We then describe several physical representations of evolving graphs that we developed (Section 4), and show results of a preliminary experimental evaluation (Section 5), which illustrates interesting performance trade-offs when different physical representations are used for an important class of analytics.

2. PORTAL BY EXAMPLE

Portal is a declarative query language for evolving graphs. We give a brief overview of the language here, see [24] for a detailed description.

Portal operates on a novel kind of a relation, called a TGraph, which can be provided as a base relation or computed as a view. A TGraph associates a sequence of consecutive non-overlapping open-closed time periods of the same duration with a sequence of snapshots. An example of a 4-snapshot TGraph is given in Figure 1, with vertex and edge relations in Figure 2.

WWW'16 Companion, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4144-8/16/04.
http://dx.doi.org/10.1145/2872518.2889290.



V

[2010, 2011)

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

[2011, 2012)

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

[2012, 2013)

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

[2013, 2014)

vid	name	salary
1	Alice	$160K
2	Bob	$100K
4	Dave	$55K
5	Eve	$90K
6	Frank	$70

E

[2010, 2011)

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

[2011, 2012)

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

[2012, 2013)

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

[2013, 2014)

vid_1	vid_2	cnt
1	2	2
1	4	2
2	4	7
5	6	5

Figure 2: Vertex and edge attributes of TGraph T1.

Portal supports unary and binary operations on TGraphs, and is fully compositional. 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.

Consider query Q1 below. This query is concise, yet it specifies a sophisticated analysis task.

```
Q1:  TSelect V [vid, pagerank()] ;
             E [vid1, vid2, sum(cnt)]
     From    T1 TOr T2
     TWhere  Start >= 2010 And End < 2014
     TGroup  by 2 years
```

Q1 combines TGraphs T1 and T2, restricts the result to the $[2010, 2014)$ range, groups the result into 2-year windows, computes pagerank() for each vertex in each time window, and sums values of the attribute cnt for each edge.

Note the use of TOr in Q1. This is one kind of a temporal join supported in Portal, returning the union of the snapshots of the two operands. (Portal also support temporal intersection TAnd, illustrated in Q2). Temporal join is a binary operation that requires its operands to be *union-compatible*. There are two parts to union-compatibility, structural and temporal. Structural union-compatibility states that vertex and edge relations of the two TGraphs must be union-compatible. Temporal union-compatibility states that temporal sequences of T1 and T2 must have the same resolution, and they must align.

As part of query Q1, Portal performs *structural aggregation*. This operation is part of temporal aggregation (TGroup) and temporal join (TOr). The default is to take the union of vertices and edges; it can be over-ridden to compute an intersection of the edges, or of the vertices, or both.

Portal supports two families of analytics. The first are snapshot analytics, which are executed on each graph in a series of temporally-adjacent snapshots. The second are trend analytics, computed across groups of temporally-adjacent snapshots. Both are illustrated in Q2 below.

```
Q2:  TSelect V [vid, trend(pr)];
             E [vid1, vid2]
     From    ( TSelect V [vid, pagerank() as pr];
                       E [vid1, vid2]
               From    T1 TAnd T2 )
     TGroup  by Size
```

Q2 executes a temporal join of T1 and T2 and invokes pagerank(), a *snapshot analytic*, on consecutive snapshots of the result. Our current focus is on Pregel-style analytics. We may accommodate additional classes in the future.

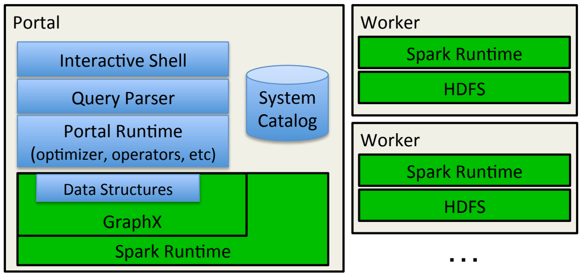


Figure 3: Portal system architecture.

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.

3. SYSTEM

The Portal system builds on GraphX [9], an Apache Spark library, as depicted in Figure 3. 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.

Query evaluation. 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 query is rewritten into a sequence of operators, and some operators are reordered to improve performance. For example, pushing temporal aggregation before temporal join can sometimes lead to better performance. A temporal join query may be rewritten to include additional temporal selection conditions, based on information about the temporal schema of the TGraphs being joined, which in turn significantly reduces data load time.

We developed several physical representations and partitioning strategies, which are selected at the physical plan generation stage. These are described in Section 4. 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 TGraph, including its temporal and structural schema.

Integration with SQL. The Portal system includes an interactive shell for exploratory data analysis. Shell users can define TGraph views, inspect query execution plans and execute SQL queries with an embedded Portal view. Consider SQL query Q3 that returns vid and tr values of 20 vertices with the most significantly increasing pagerank trend.

```
Q3:   Select    VF.vid, VF.tr
      From      T5.toVerticesFlat() as VF
      Order by  tr
      Limit     20
```

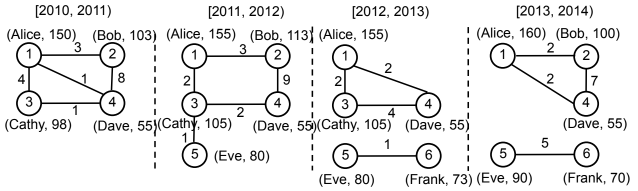


Figure 4: SG representation of T1 from Figure 1.

An important part of Q3 is the use of T5.toVerticesFlat() in the From clause. This is an operation provided by the Portal framework, which collects all vertices in the union of snapshots of T5 into a single nested vertex collection and flattens it into VF (vid:int, start:date, end:date, tr:float, mx:float). VF can be used in SQL queries. Portal also provides an operation that returns a flattened collection of edges, called toEdgesFlat().

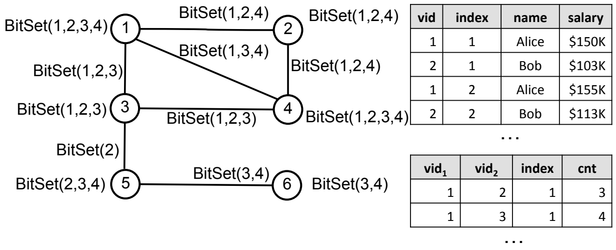
4. PHYSICAL REPRESENATATIONS

We considered three in-memory TGraph representations that differ in compactness, but also, perhaps more importantly, in the kind of locality they prioritize. With *structural locality*, neighboring vertices of the same snapshot are laid out together, while with *temporal locality*, consecutive states of the same vertex are laid out together. SnapshotGraph (SG), a representation in which each snapshot is stored explicitly, naturally preserves structural locality, but temporal locality is lost. OneGraph (OG) stores all vertices and edges of an evolving graph once, in a single data structure. This representation emphasizes temporal locality, while also preserving structural locality. HybridGraph (HG) trades compactness for better structural locality, by aggregating together several consecutive snapshots, and computing a OneGraph for each snapshot cluster.

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 4. 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 [17]. 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.

OneGraph (OG). The most topologically compact representation of graph structure 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. The drawback is that OG is much denser than individual snapshots of SG. OG stores vertex and edge attribute information 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.



vid	index	name	salary
1	1	Alice	$150K
2	1	Bob	$103K
1	2	Alice	$155K
2	2	Bob	$113K

...

vid_1	vid_2	index	cnt
1	2	1	3
1	3	1	4

...

Figure 5: OG representation of T1 from 1.

HybridGraph (HG). As an intermediate representation between SG and OG, we implement the HybridGraph (HG) data structure. HG is a series of OGs, with each OG representing some number of temporally adjacent snapshots.

In our current implementation each OG in the sequence corresponds to the same number of temporally adjacent snapshots. This is the simplest snapshot clustering method, yet, as we will see in Section 5, it already improves performance compared to OG. However, we also observed that placing the same number of snapshots into each cluster often results in unbalanced cluster sizes. This is because networks commonly exhibit strong temporal skew, with later snapshots being significantly larger than earlier ones. Consequently, we are currently working on more sophisticated clustering approaches that would lead to better balance, and ultimately to better performance.

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.

In our experiments we compare performance of SG, OG and HG with (1) no repartitioning after load, and (2) with repartitioning using the 2D edge partitioning strategy (E2D). This strategy is available in GraphX and was used without modification. 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. As we will see in Section 5, E2D provides good performance for Pregel-style analytics.

We implemented and are experimenting with additional partitioning strategies as part of our ongoing work.

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.

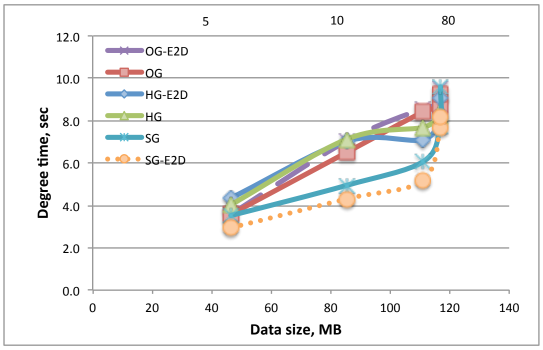


Figure 6: Degrees time, DBLP.

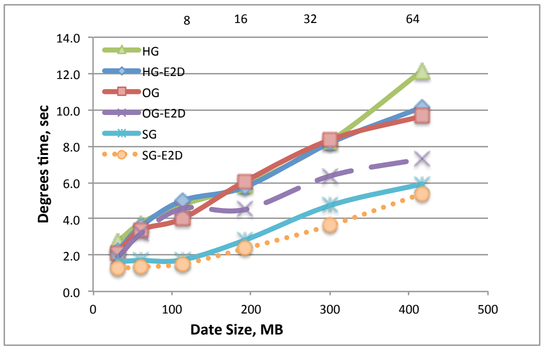


Figure 7: Degrees time, nGrams.

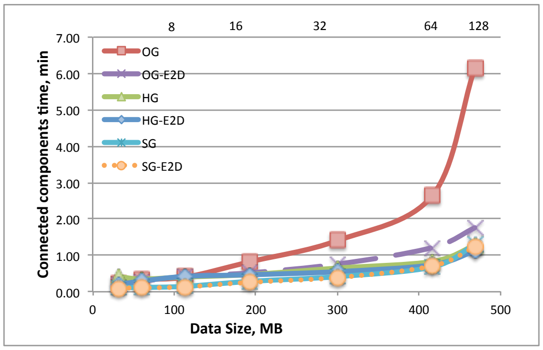


Figure 8: Connected Components, nGrams.

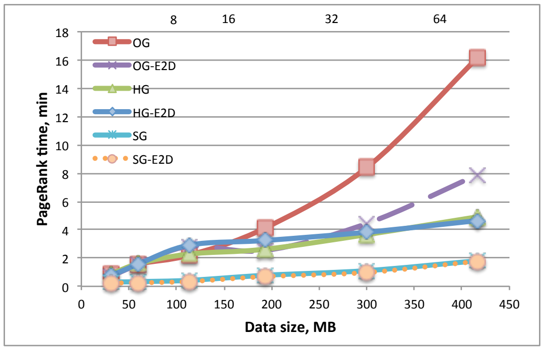


Figure 9: PageRank, nGrams.

Data. We evaluate performance of our framework on two real open-source datasets. DBLP[1] is a 250 MB dataset that contains co-authorship information from 1936 through 2015, with over 1.5 million author nodes and over 6 million undirected co-authorship edges. nGrams[2] is a 40 GB dataset that contains word co-occurrences from 1520 through 2008, with over 1.5 million word nodes and over 65 million undirected co-occurrence edges.

The nGrams dataset is of comparable size to the LiveJournal dataset in [23] 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. We plan to experiment with a larger DELIS dataset [5] as we grow our cluster in the near future.

Degrees. Computation of the number of edges for each vertex is performed using a single round of messages between nodes, with batch mode for HG and OG. We used the following query to evaluate data structure performance over varying number of snapshots:

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

Results of this experiment are presented in Figures 6 and 7, both with warm start. Observe that SG outperforms HG and OG for smaller data sizes. This is contrary to our expectation that batch mode of HG and OG would always be faster than SG. SG performance can be explained if we consider that each snapshot is spread out over fewer partitions than in the aggregate data structures. Thus, more communication occurs intra-partition rather than between partitions, which in turn dominates the overall running time. Furthermore, we expected HG performance to be between SG and OG, the two data structures that it combines. We do observe this in most cases, but not consistently. We believe that HG does not consistently outperform OG due to its sensitivity to temporal skew. This effect is particularly pronounced for fast-running operations like Degree.

Connected components. Snapshot analytics like Connected Components are implemented using the Pregel API in GraphX, with batch mode for HG and OG. The algorithm was executed until convergence, with no limit to the number of iterations. We used the following query to evaluate data structure performance over varying number of snapshots:

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

SG performs better than the other data structures in this experiment, contrary to our expectation that batch mode of HG and OG would be faster (Figure 8). This can be explained by HG and OG 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. Iterations get faster as vertex values converge, and vertices stop sending messages. In OG/HG, a vertex converges only when it does so in all represented snapshots.

However, note that as the number of total snapshots increases, HG performance improves compared to SG, and in

[1] http://dblp.uni-trier.de/

[2] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

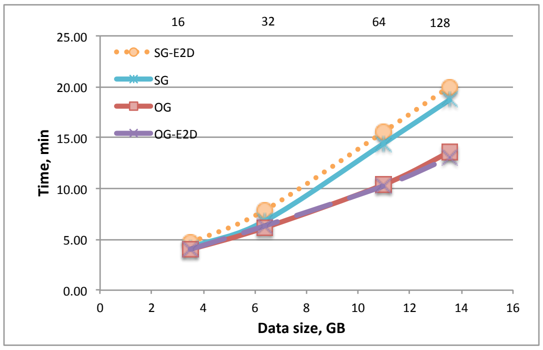


Figure 10: trend(degrees()).

fact for the largest size (128 snapshots) HG surpasses SG in performance. We saw this trend in both data sets.

PageRank. Like Connected Components, PageRank is implemented using Pregel. The query in this experiment is the same, replacing components with pagerank. PageRank was executed for 10 iterations or until convergence, whichever came first. Results of this experiment (Figure 9) are similar to those of Connected Components. SG outperforms the other data structures, but HG exhibits the same slope and its performance improves relative to the other data structures as the number of snapshots is increased. E2D partitioning leads to performance improvements for SG, but inconsistent for HG/OG.

Mixed queries. All the experiments so far evaluated performance of individual Portal snapshot analytics operations. Our final experiment considers performance of snapshot and trend analytics that are executed as part of a query that also includes temporal selection (TWhere) and aggregation (TGroup). Note that we do not currently have an implementation of temporal aggregation for HG, and so HG is not used in this experiment.

```
TSelect V[vid, trend(deg)]; E
From    (TSelect V[vid, degrees() as deg]; E[vid1, vid2]
         From    nGrams
         TWhere  Start >= :x And End < :y)
TGroup  by 16 years
```

This query computes the degree of each vertex in each snapshot, aggregates snapshots in groups of 16, and uses trend as the aggregation function on degrees. We have shown above that in most cases SG provides the most efficient performance for snapshot analytics. We have also shown in [24] that aggregate data structures (OG, HG, others) take longer to load but are more efficient for TGroup. Figure 10 (warm start) shows that, when these operations are combined, OG outperforms SG. Temporal aggregation is a more expensive operation in this scenario that the degrees analytic.

We conclude this section with a cold-start execution of the following SQL-Portal query:

```
Select vid, pr
From (TSelect V[vid, trend(prank) as pr];  E
      From (TSelect All V[vid, pagerank() as prank];
                    All E
            From nGrams
            TWhere Start >= :x And End < :y
            TGroup by 16 years)
     TGroup by size).toVerticesFlat()
Order by pr
Limit 10
```

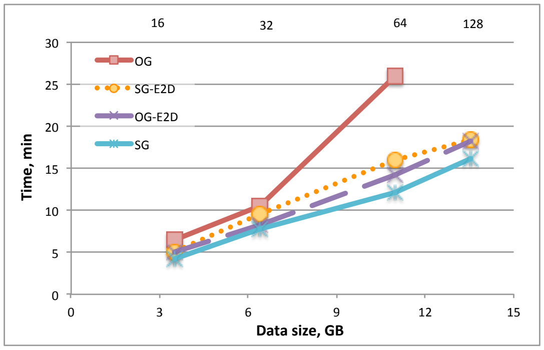


Figure 11: TGroup, PageRank, trend.

Here, SG with no partitioning, and OG with E2D show comparable performances, as seen in Figure 11.

In summary, running times of the best-performing data structure for each experiment are reasonable, and show a linear trend with increasing data size. Interestingly, no one data structure is most efficient across all operations. SG is most efficient for snapshot analytics in most cases, but HG outperforms it for larger data sizes, while OG provides a good balance for mixed queries. These insights motivate future work on query optimization in Portal.

6. RELATED WORK

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. [10] 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. [21] focus on historical reachability queries.

Our work shares motivation with Miao et al. [17], 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.

Temporal SQL. Like Temporal SQL [13], Portal assigns temporal meaning to data. To that effect, we use the definition of a time period from the SQL:2011 standard, and support temporal predicates including overlaps, precedes and contains in the TWhere clause. Unlike Temporal SQL, Portal explicitly incorporates the temporal dimension into operations, including language primitives and aggregate functions, both built-in and user-defined. Specifically, Temporal SQL does not support period aggregates, so expressions like TGroup by 2 years would require new user-defined functions on periods. Additionally, Portal aggregation functions are inherently time-centric on the level of individual vertex and edge attributes, which allows for such aggregates as first, last, and trend. Another important difference between Temporal SQL and Portal is that Portal is graph-centric. Our language natively supports snapshot analytics, which operate on vertex and edge relations in their entirety. Portal users can register their own snapshot analytics through the User-Defined Analytics API.

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 [11, 12, 17].

Khurana and Deshpande [11] 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 OneGraph.

Ren et al. [19] develop an in-memory representation of evolving graphs based on representative graphs for sets of snapshots. Our OneGraph can be thought of as a representative graph for the whole selected time period, while HybridGraph is a sequence of representative graphs.

Semertzidis et al. [21] 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.

Boldi et al. [4] present a space-efficient non-delta approach for storing a large evolving Web graph that they harvested. Their work represents purely topological information and does not address vertex and edge attributes.

Distributed frameworks. We build upon GraphX [9], 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. Lin et al. [16] proposed Warcbase, an infrastructure for exploration and discovery of Web archives built on HBase. Their work shares motivation with ours, but the technical choices are fairly different from ours. Most importantly, we work with Apache Spark, a higher-level abstraction over MapReduce, and define a declarative query language on top of Apache Spark.

7. CONCLUSIONS AND FUTURE WORK

In this paper we gave an overview of Portal, a declarative query language for evolving graphs. We described an implementation of Portal in scope of Apache Spark, a distributed open-source processing framework. Our main focus here was on presenting several physical representations of evolving graphs, and on experimentally evaluating their their relative performance for several interesting analytics.

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 their performance.

8. REFERENCES

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

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

[3] 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.

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

[5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, 2004.

[6] J. Chan, J. Bailey, and C. Leckie. Discovering correlated spatio-temporal changes in evolving graphs. *Knowl. Inf.Syst.*, 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] M. Goetz, J. Leskovec, M. McGlohon, and C. Faloutsos. Modeling blog dynamics. In *ICWSM*, 2009.

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

[10] A. Kan, J. Chan, J. Bailey, and C. Leckie. A Query Based Approach for Mining Evolving Graphs. In *AusDM*, 2009.

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

[12] G. Koloniari. On Graph Deltas for Historical Queries. Istanbul, Turkey, 2012.

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

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

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

[16] J. Lin, M. Gholami, and J. Rao. Infrastructure for supporting exploration and discovery in web archives. In *WWW*, 2014.

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

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

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

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

[21] K. Semertzidis, K. Lillis, and E. Pitoura. TimeReach: Historical Reachability Queries on Evolving Graphs. In *EDBT*, 2015.

[22] 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.

[23] 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.

[24] V. Zaychik Moffitt and J. Stoyanovich. Portal: A Query Language for Evolving Graphs. *ArXiv e-prints*, Feb. 2016. arXiv:1602.00773.