Framework for Querying and Analysis of Evolving Graphs

A Thesis

Submitted to the Faculty

of

Drexel University

by

Vera Zaychik Moffitt

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy in Information Studies

May 2017

Dedications

To all working mamas.

And to my family, for everything.

"Maybe there weren't any grown-ups, only people who had worked hard and also got lucky and were slightly out of their depth, all of us doing the best job we could, which is all we can really hope for."

Neil Gaiman

Acknowledgements

I would like to express a sincere and heartfelt gratitude to my advisor Dr. Julia Stoyanovich. The path of graduate research is never smooth or straight, but her encouragement and participation made this work possible. She took a chance on a part-time student with a full-time job and two small children and I hope I have not let her down. The work contained in this dissertation is truly a result of shared labor between us over the past several years. Any errors contains herein are, of course, solely mine.

I would also like to thank Dr. William C Regli, whom I have had the privilege to know for almost 20 years and who not only encouraged me to finally pursue a doctorate degree, but went to bat for me with the acceptance committee.

Lockheed Martin enabled me to focus on my graduate studies by means of a special Doctoral Leadership program, and I am particularly thankful to Scott Fouse, William Borgia, Dr. David Pustai, and Jeff Wilcox for nominating, accepting, and supporting me in this program over the last three years.

The Database group was my home and I had the opportunity to work with great undergraduate and graduate students there. Special mention to Shishir Kharel, Halima Olapade, and Amir Pouya Agha Sadeghi.

My friends played a large role in encouraging and supporting me. In particular, I would like to thank Jerry Franke and Dr. Lisa Anthony, both of whom, having experienced graduate life before, shared their wisdom. It does help to know that every graduate student goes through these ups and downs.

Finally, I would like to express at least the small degree of gratefulness and appreciation to my family. To Amy D'Apice, for encouraging me to pursue my doctorate while being a mother and serving as the amazing inspiration. To Rita Moffitt, my wonderful mother-in-law, who took up the slack of child pickups and bedtimes with the utmost love and support. To my mother Galina Voronina and father Pavel Zaychik, both of whom had not the least bit of doubt about my ability to bring this effort to fruition and were always on my side through the inevitable rejections and setbacks. To my two amazing sons, for being so understanding and supportive when I made myself scarce to work, and for giving me so much joy. And to my loving husband Craig Moffitt, for talking me off ledges, and being there, for better or worse.

Table of Contents

List of Tables . . . viii
List of Figures . . . x
Abstract . . . xiv
1 Introduction . . . 1
1.1 Existing Models . . . 2
1.2 Property Graph Model and RDF . . . 5
1.3 Motivating Examples . . . 6
1.3.1 Node Influence over Time . . . 7
1.3.2 Graph Centrality over Time . . . 7
1.3.3 Communities over Time . . . 9
1.3.4 Spread of Information . . . 10
1.4 Contributions of this Dissertation . . . 10
1.5 Structure of this Document . . . 11
2 Background . . . 12
2.1 Temporal Relational Model . . . 12
2.1.1 Relations . . . 12
2.1.2 Temporal Predicates . . . 15
2.1.3 Temporal Relational Algebra, TRA . . . 17
2.1.4 Nested Temporal Relational Algebra . . . 22
2.2 Graph Model . . . 23
2.2.1 Property Graph Model . . . 23
2.2.2 Graph Queries . . . 26
3 Evolving Graphs Model . . . 39
3.1 TGraph – Temporal Graph Model . . . 39
3.2 Temporal Graph Algebra . . . 45
3.2.1 Trim . . . 47
3.2.2 Map . . . 48
3.2.3 Subgraph . . . 50
3.2.4 Aggregation . . . 53
3.2.5 Node Creation . . . 54

3.2.6 Union . . . 58
3.2.7 Intersection . . . 61
3.2.8 Difference . . . 62
3.2.9 Edge Creation . . . 63
3.3 Usecases . . . 65
3.3.1 Vertex Influence over Time . . . 65
3.3.2 Graph Centrality over Time . . . 66
3.3.3 Communities over Time . . . 67
3.3.4 Spread of Information . . . 68
3.4 Formal TGA properties . . . 68
3.4.1 Temporal Groupedness . . . 69
3.4.2 Temporal Completeness . . . 70
3.4.3 Expressive Power . . . 75
4 System . . . 77
4.1 Background . . . 77
4.2 Architecture . . . 80
4.3 Physical Layout . . . 82
4.3.1 Locality . . . 82
4.3.2 Snapshot Groups . . . 83
4.4 In-Memory Representations . . . 86
4.4.1 VertexEdge . . . 87
4.4.2 RepresentativeGraph . . . 93
4.4.3 OneGraph . . . 93
4.4.4 HybridGraph . . . 94
4.5 Optimizations . . . 95
4.5.1 Lazy Coalescing . . . 95
4.5.2 Lazy Referential Integrity Enforcement . . . 97
4.5.3 Trim Pushdown . . . 99
4.6 Limitations . . . 99
5 Evaluation . . . 101
5.1 Setup . . . 101
5.2 Physical layout . . . 104

5.2.1 Local Point Queries . . . 105
5.2.2 Local Range Queries . . . 106
5.2.3 Global Point Queries . . . 108
5.3 Optimizations . . . 109
5.3.1 Lazy Coalescing . . . 109
5.3.2 Filter Pushdown . . . 110
5.4 In-Memory Representations . . . 111
5.4.1 Trim . . . 111
5.4.2 Map . . . 113
5.4.3 Vertex Subgraph . . . 113
5.4.4 Aggregation . . . 115
5.4.5 Node Creation . . . 120
5.4.6 Binary Operations . . . 121
5.5 Comparison to Baseline . . . 127
5.5.1 GStar as a Baseline . . . 127
5.5.2 ImmortalGraph as a Baseline . . . 135
5.6 Usecases . . . 139
6 Related Work . . . 142
6.1 Evolving Graph Models . . . 142
6.2 Evolving Graph Queries . . . 144
6.2.1 Temporal operators . . . 145
6.2.2 Nontemporal operators . . . 148
6.3 Mapping . . . 149
6.4 Nontemporal Graph Languages . . . 149
6.4.1 StruQL . . . 150
6.4.2 SocialScope . . . 150
6.4.3 GraphLog . . . 151
6.5 Evolving Graph Systems . . . 152
7 Future Work . . . 155
7.1 TGA with Sequenced Semantics . . . 155
7.1.1 Critique of TGraph Model . . . 155
7.1.2 TRA with Sequenced Semantics . . . 157

7.1.3 TGraph Model with Sequenced Semantics 157
7.1.4 Sequenced Semantics in a Distributed Environment 158
7.2 Declarative Language 160
7.3 Query Optimization 163
8 Conclusions 165
Bibliography 167

List of Tables

2.1 Example temporal relation instance $\mathbf{r}$. 13

2.2 Uncoalesced instance of relation $\mathbf{r}'$. 15

2.3 Example temporal relation instance $\mathbf{r}$. 17

2.4 Example temporal relation instance $\mathbf{s}$. 17

2.5 $\pi^T_N(\mathbf{r})$ 18

2.6 ${}_S\gamma^T_{count(N)}(\mathbf{r})$ 19

2.7 $\mathbf{r} \times^T \mathbf{s}$ 20

2.8 Example temporal relation instance $\mathbf{v}$. 21

2.9 $\mathbf{r} \cup^T \mathbf{v}$ 21

2.10 $\mathbf{r} \cap^T \mathbf{v}$ 21

2.11 $\mathbf{r} \setminus^T \mathbf{v}$ 21

2.12 Example nested temporal relation instance $\mathbf{n}$. 22

2.13 $\mu^T_C(\mathbf{n})$ 23

2.14 Properties of graph G. 24

2.15 Set of bindings of variables of P_1 in G. 27

2.16 Set of bindings of variables of P_2 in G. 28

3.1 A co-authorship network represented using the TGraph model, consisting of two nested temporal relations. 39

3.2 Example nested temporal relation $\mathbf{w}$. 47

3.3 $\mathcal{R}(max(S), count(C), \mathbf{w})$ 47

3.4 Example TGraph $\mathcal{G}$. 47

3.5 $\mathsf{trim}^T_{[2015/5,2015/8)}(\mathcal{G})$ 48

3.6 $\mathsf{map}^T_v(\pi_{type,name}, \mathcal{G})$ 49

3.7 map^T_e(cnt as count when p overlaps [2015/3,2015/6), $\mathcal{G}$) 49

3.8 $\mathsf{subgraph}^T$ with pattern P_1 returns a graph with no isolated nodes. 50

3.9 Subgraph of nodes with a school property connected by edges that persist for longer than 2 months. . . . 52

3.10 $\mathsf{agg}^T(p_2, G)$ . . . 53

3.11 Attribute-based node creation based on property school. . . . 55

3.12 Node creation with a 3-month window. . . . 56

3.13 $\mathsf{node}^T_w(r_v = \mathsf{always}, r_e = \mathsf{exists}, f_{v_1} = \mathsf{first(name)}, f_{v_2} = \mathsf{first(school)}, \mathcal{G})$ . . . 58

3.14 Example TGraph $\mathcal{G}_2$. . . . 59

3.15 Union of $\mathcal{G}$ and $\mathcal{G}_2$. . . . 59

3.16 Example TGraph $\mathcal{G}_3$. . . . 60

3.17 $\mathcal{G} \cup^{TG}_{\mathsf{left(name),max(count)}} \mathcal{G}_3$ . . . 61

3.18 $\mathcal{G} \cap^{TG}_{\mathsf{left(name),max(count)}} \mathcal{G}_3$ . . . 62

3.19 $\mathcal{G} \setminus^{TG} \mathcal{G}_3$ . . . 62

3.20 Result of edge creation operator over $\mathcal{G}$ and $\mathcal{G}_2$. . . . 63

3.21 $\mathsf{edge}^T_P(\mathcal{G}, \mathcal{G}_2)$ . . . 65

4.1 A co-authorship network from Table 3.1, reproduced here for convenience. . . . 84

4.2 TGA operators in each in-memory representation. . . . 88

4.2 TGA operators in each in-memory representation, continued. . . . 89

4.2 TGA operators in each in-memory representation, continued. . . . 90

5.1 Experimental datasets. . . . 101

5.2 In-degree centrality over time in wiki-talk . . . 140

6.1 Classification of temporal graph queries, according to Miao et al. [74] . . . 144

6.2 Mapping between published query types and TGA operators. . . . 149

6.3 Mapping between TGA and other published systems. . . . 154

7.1 A social network as coalesced temporal relations. . . . 156

List of Figures

1.1 Partial map of the WWW in 2003. Each node is an IP address and the edges are the routes between them. Credit: Creative Commons The Opte Project 2

1.2 The protein interaction network of T. pallidum including 576 proteins and 991 interactions. ©Titz et al. [87] 2

1.3 Evolution of Apple's inventor network over a 6-year period. ©Vermeij, André. [90]. . 3

1.4 Time-varying Social Network Data Model [20]. 4

1.5 A property graph model with periods of validity represented by a new time property. Nodes are solid line boxes, while edges are dashed line boxes. 5

1.6 Wiki-talk graph centrality over time. The centrality measure is a ratio of the sum of centrality of each node to the maximum possible, a star graph. Wikitalk is a sparse, loosely connected graph. As the size of the graph increases, centrality decreases sharply. 8

1.7 Wiki-talk connected components using 1-year temporal resolution. 9

2.1 Snapshot Reducibility maps each temporal operator to its nontemporal counterpart. Based on [17], Figure 3. 14

2.2 X overlaps Y 15

2.3 X contains Y 16

2.4 X meets Y 16

2.5 X precedes Y 16

2.6 X succeeds Y 16

2.7 Example graph G of a social network. 24

2.8 Example graph G depicted with property sets. Type of all people nodes, i.e., Julia, John, etc., is not shown for readability. 25

2.9 Example basic pattern P_1. 26

2.10 Example basic pattern P_2. 28

2.11 subgraph(G,P_1) 29

2.12 subgraph(G,P_2) 29

2.13 Example navigational graph pattern P_3. 30

2.14 subgraph(G,P_3). Properties of the people nodes not shown for readability. 30

2.15 Example navigation graph pattern P_4. 30

2.16 Example navigational graph pattern with aggregation, P_5. 32

2.17 Example navigational graph pattern with aggregation, P_6. 32

2.18 Example navigational graph pattern with aggregation, P_7. 32

2.19 Example NGP with Skolemization, P_8. 34

2.20 $\mathsf{node}(G, P_8)$. New nodes and edges are highlighted in orange. Properties of the people nodes are not shown for readability. 35

2.21 Example NGP with Skolemization, P_9. 36

2.22 Example join of two graphs G_1 and G_2. 36

2.23 Example composition of two graphs G_1 and G_2 with directional condition $\sigma_1 = dst$, $\sigma_2 = src$. 38

3.1 Snapshot sequence representation of the example TGraph from Table 3.1. Node and edge type property ommitted for readability. 40

3.2 Navigational graph pattern P_1, restricting to nonisolated nodes only. 50

3.3 Temporal navigational graph pattern restricting nodes by period of validity. 51

3.4 TNGP p_2 that computes the degree of each node accounting only for old edges. . . . 53

3.5 TNGP to create nodes for each value of school property. 54

3.6 Visualization of edge creation over $\mathcal{G}$ and $\mathcal{G}_2$ using snapshots. 64

3.7 Temporal navigational graph pattern p_1 to compute node degrees. 65

3.8 Navigational graph pattern p_2. 66

3.9 Connected components pattern p_3. 67

3.10 A journey pattern p_7. 68

4.1 Apache Spark task processing architecture. ©Arush Kharbanda. [56]. 78

4.2 Distributed representation of a graph, Fig.3 in [44]. 78

4.3 Portal system architecture. 80

4.4 Tuple intervals are split into four snapshot groups. 83

4.5 Tuples of TV and TE in TGraph from Table 4.1, split into three equi-depth groups. . 85

4.6 OG representation of a small TGraph. 93

5.1 The wiki-talk dataset exhibits a quadratic relationship between the number of nodes and edges. 102

5.2 The size of the largest connected component in the wiki-talk dataset. 102

5.3 The graph clustering coefficient of the wiki-talk dataset over time. 102

5.4 The edge reciprocity of the wiki-talk dataset over time. 102

5.5 The nGrams dataset exhibits a linear relationship between the number of nodes and edges. . . . 103

5.6 The size of the largest connected component in the nGrams dataset. . . . 103

5.7 The graph clustering coefficient of the nGrams dataset over time. . . . 103

5.8 The edge reciprocity of the nGrams dataset over time. . . . 103

5.9 The Twitter dataset exhibits a linear relationship between the number of nodes and edges. . . . 104

5.10 The size of the largest connected component in the Twitter dataset. . . . 104

5.11 Performance of each snapshot group and locality format on local point queries on the wiki-talk dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method. . . . 105

5.12 Performance of each snapshot group and locality format on local point queries on the nGrams dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method. . . . 105

5.13 Performance of each snapshot group and locality format on local range queries on the wiki-talk dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method. . . . 107

5.14 Performance of each snapshot group and locality format on local range queries on the nGrams dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method. . . . 107

5.15 Performance of each snapshot group and locality format on global point queries. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method. . . . 108

5.16 Performance of the trim operation with eager and lazy coalescing. . . . 109

5.17 Performance of the map operation with eager and lazy coalescing. . . . 110

5.18 Performance of the trim operation with filter pushdown. . . . 110

5.19 Performance of trim operator over varying intervals. . . . 112

5.20 Performance of vertex-map operator over varying intervals on wiki-talk. . . . 113

5.21 Performance of vertex-subgraph operator over varying selectivity. . . . 114

5.22 Performance of aggregation operator over varying intervals, computing node degrees. $\mathsf{agg}_p^T(\mathsf{trim}_t^T(\mathcal{G}))$ . . . 116

5.23 Performance of snapshot analytics over varying time intervals on the wiki-talk dataset. 117

5.24 Performance of snapshot analytics over varying time intervals on the nGrams dataset. 118

5.25 Performance of snapshot analytics over varying time intervals on the Twitter dataset. 119

5.26 Performance of attribute-based node creation on the wiki-talk dataset with varying temporal window, over the edit count node property. . . . 121

5.27 Performance of temporal-window node creation on the wiki-talk dataset with varying window size. $\mathsf{node}_w^T(\mathsf{w} = \mathsf{w}_1, \mathsf{r_v} = \mathsf{always}, 'wiki - talk')$ 122

5.28 Performance of temporal-window node creation on the nGrams dataset with varying window size. $\mathsf{node}_w^T(\mathsf{w} = \mathsf{w}_1, \mathsf{r_v} = \mathsf{always}, 'nGrams')$ 123

5.29 Performance of the union operator with varying temporal overlap over the same dataset. 124

5.30 Performance of the intersection operator with varying temporal overlap over the same dataset. 125

5.31 Performance of the difference operator with varying temporal overlap over the same dataset. 126

5.32 Performance of G* and Portal on the average node degree query. 129

5.33 Performance of G* and Portal on the clustering coefficient distribution query. 132

5.34 Performance of G* and Portal on the minimum distance distribution query. 133

5.35 Performance of G* and Portal on the connected component size distribution query. . 135

5.36 Performance of ImmortalOneGraph (IOG) on snapshot analytics on the wiki-talk dataset. 136

5.37 Performance of ImmortalOneGraph (IOG) on snapshot analytics on the nGrams dataset. 137

5.38 Performance of ImmortalOneGraph (IOG) on snapshot analytics on the Twitter dataset. 138

5.39 In-degree centrality with 1 year resolution. 140

5.40 Communities with 1 year resolution. 141

6.1 Example snapshot sequence representing graph evolution over six stages. 143

6.2 An example GraphLog query: the descendants of P1 which are not descendants of P2. [31], Figure 2, p.407 152

7.1 A social network as a snapshot sequence. 155

7.2 Vertices from Figure 7.1 split in 4 partitions. 158

Abstract

Framework for Querying and Analysis of Evolving Graphs
Vera Zaychik Moffitt
Julia Stoyanovich, Ph.D. and William C. Regli, Ph.D.

Graph representations underlie many modern computer applications, capturing the structure of such diverse networks as the Internet, personal associations, roads, sensors, and metabolic pathways. While the static structure of graphs is a well-explored field, a new emphasis is being placed on understanding and representing the way these networks change over time. Current research is delving into graph evolution rate and mechanisms, the impact of specific events on network evolution, and spatial and spatiotemporal patterns. However, systematic support for scalable querying and analytics over evolving graphs still lacks.

In this dissertation, we combine the theoretical and practical advances in graph databases and temporal relational databases to formulate an evolving graph model, including a representation and an algebra. With this model we aim to enable systematic support for the evolving graph analysis, heretofore lacking in generality. Our goal is to give users an ability to concisely express a wide range of common analysis tasks. We provide several use cases to motivate our work, and demonstrate how they are supported in our model. We show that our algebra is strictly more expressive than the currently published state of the art and provides additional operations not available to the users today.

We also provide a prototype implementation of our model in a distributed system called Portal and conduct an extensive experimental evaluation with real datasets. The results of our experiments show that Portal scales to large evolving graph datasets even on a modest size cluster and outperforms a published baseline, while providing additional functionality not available elsewhere.

Chapter 1: Introduction

Many social structures and systems can be represented as networks or graphs, the two terms we use interchangeably here. The World Wide Web is often thought of as a graph where Web pages or Web sites are the graph nodes, and the hyperlinks between them are the graph edges (Figure 1.1). A social network is another well-known graph type, with people or organizations as nodes and their activities or relationships as edges. Graphs can be composed based on sensor and road networks, animal herds (for the purpose of studying epidemics), metabolism pathways, protein interaction networks (Figure 1.2) and many others.

Considerable research and engineering effort is being devoted to developing effective and efficient graph representations and analytics. Efficient graph abstractions and analytics for *static graphs* are available to researchers and practitioners in scope of open source platforms such as Apache Giraph, Apache Spark / GraphX [44] and GraphLab / PowerGraph [43].

The phenomena that are represented by these graphs can change over time, some continuously and others sporadically. The Web of a year ago is quite different than the Web of today. According to some estimates, half of the Web is replaced every 50 days [26]. The Apple internal Innovation Network doubled in size between 2009 and 2012 [90], as can be seen in Figure 1.3. Many interesting questions about these networks are related to their evolution rather than their static state. Researchers study graph evolution rate and mechanisms, e.g., [4, 26]), impact of specific events on further evolution, e.g., [22]), spatial and spatio-temporal patterns, e.g., [63]), with most progress taking place in the last decade [55, 74, 77, 81]. Some areas where evolving graphs are being studied are social network analysis [42, 66, 67, 80], biological networks [10, 11, 85] and the Web [23, 76].

At the same time, considerable research has been undertaken in the area of temporal relational databases since the 1980s. So much so that a large portion of the definitive Encyclopedia of Database Systems [70] is dedicated to temporal items. This work includes how to represent time and temporal tuples [30, 54, 83], semantics of different temporal models [12], temporal algebra [35], and access methods [79]. The SQL:2011 standard provides partial support for temporal data [61].

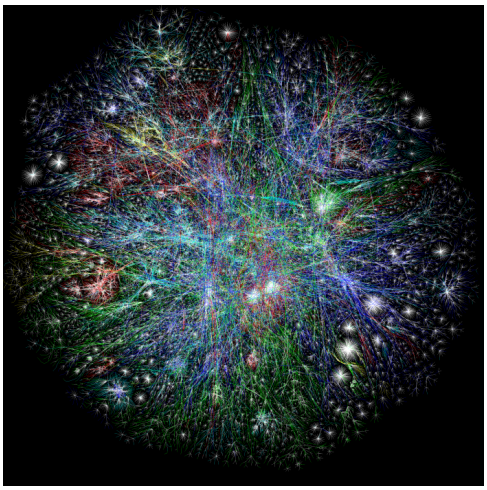


Figure 1.1: Partial map of the WWW in 2003. Each node is an IP address and the edges are the routes between them. Credit: Creative Commons The Opte Project

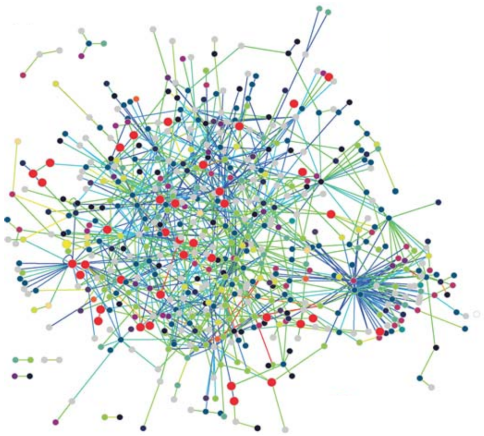


Figure 1.2: The protein interaction network of T. pallidum including 576 proteins and 991 interactions. ©Titz et al. [87]

Given the progress both in the area of graph databases and temporal databases, it is expected that evolving graphs would be supported as well, being at the intersection of the two areas. And yet, systematic support for scalable querying and analytics over evolving graphs is still lacking, despite much recent interest and activity and despite increased variety and availability of evolving graph data. This support is urgently needed, due to the scalability and efficiency challenges inherent in evolving graph analysis, and to considerations of usability and ease of dissemination. *The goal of our work is to fill this gap by combining the advances in graph databases on the one hand and temporal relational databases on the other.*

1.1 Existing Models

The need to query evolving graphs has been recognized for some time. Five different properties must be considered when extending a model and a language with temporal information:

- Correctness. Queries should have the expected and correct semantics. The time information presents a new dimension to the existing data, and care must be taken to preserve temporal integrity of the query results.
- Expressivity. Many classes of evolving graph queries exist, as we review in Section 6.2. It is desirable that the majority or all of these classes are covered by the approach.
- Usability. The queries must be reasonably easy to formulate for an average user. The user must not have to work too hard to use the language. If a basic query is too difficult to

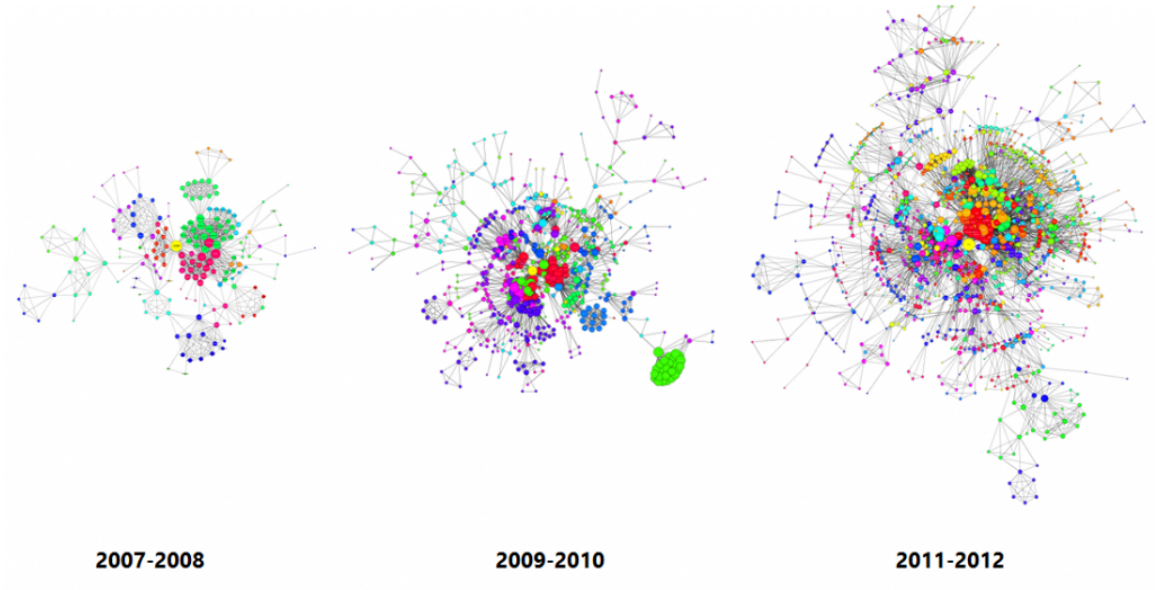


Figure 1.3: Evolution of Apple's inventor network over a 6-year period. ©Vermeij, André. [90].

compose even for people with extensive programming experience, then the language is not usable and many mistakes will be made in its use.

- Upward compatibility. To facilitate transition for existing users of static graph analysis systems, it is desirable that the temporal extension is syntactically similar and the semantics of its operations in the basic case is the same as applying static analysis at each point in time. There are several other aspects of upward compatibility and we return to them in Section 3.4, when we analyze the formal properties of our model.
- Performance. Graph evolution over months and years is commonplace – we have three publicly available datasets that span years – and the model, as well as any system implementation, must scale both with the graph size dimension and with the time dimension.

It is tempting to use the existing graph models for the purpose of representing evolving graphs by adding time as data. Consider Figure 1.4, where an evolving interaction network is represented as a sequence of frames within which actors have interactions, using the Neo4j database. Frames are connected to timelines to represent time granularity (not depicted). This approach is one example of *time as data*, and here specifically the temporal information is introduced through addition of temporal nodes and linking of actors to those nodes through special temporal edges.

There are several issues with this approach, namely, of expressivity, usability, and performance. With regard to expressivity, it is not clear how a spatio-temporal pattern can be used to query this graph. Which actors have repeated interactions over a period of interest? What are the stable

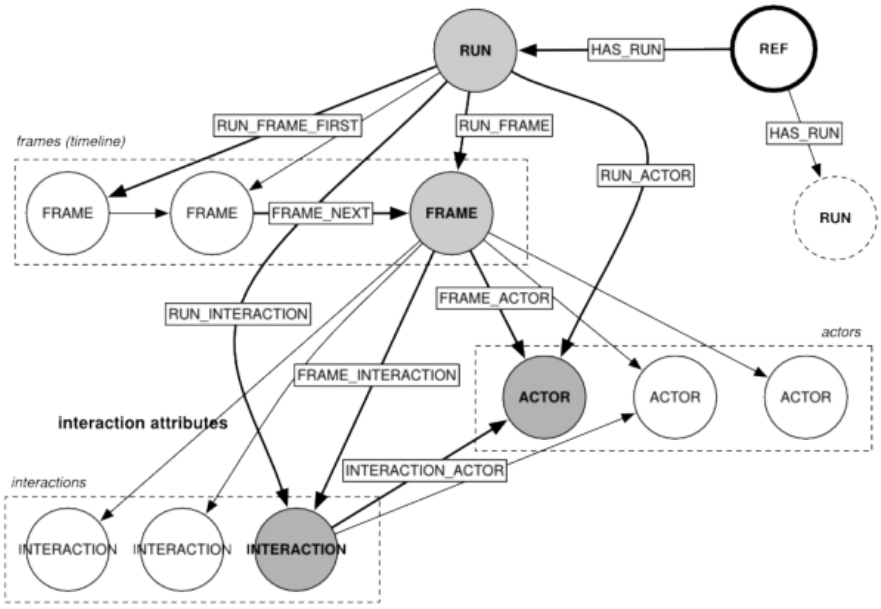


Figure 1.4: Time-varying Social Network Data Model [20].

connected components? What anomalies can be found, where an anomaly is a temporal pattern of behavior?

Usability-wise, the queries over such graphs are difficult to formulate and difficult to understand. For example, a Cypher query "List all distinct days on which an actor was present." is expressed as follows [20]:

```
START actor = node(some_actor_id)
MATCH ()-[d:NEXT_LEVEL]->()-[:NEXT_LEVEL]->timeline,
    timeline-[:TIMELINE_INSTANCE]-()-[:FRAME_ACTOR]-actor
RETURN DISTINCT(d.day)
```

This query traverses arbitrary time dimension levels and requires an understanding of the frame and run hierarchy. All time information is represented indirectly. We would be hard-pressed to query this model effectively.

Finally, as the experiments show, the resulting graph is extremely dense even for a small number of nodes as the time dimension is extended [20]. As a result, query performance degrades and quickly becomes too slow to support interactivity.

Another approach is to add time information as another property in a node or an edge within the property graph model. If we allow nodes to have varying periods of validity, i.e., appear and disappear as needed, then the time property must be a set of times or periods. Even so, time as

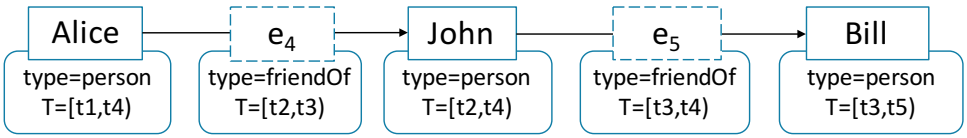


Figure 1.5: A property graph model with periods of validity represented by a new time property. Nodes are solid line boxes, while edges are dashed line boxes.

a property does not allow node properties to change over time. Ignoring this issue for the moment, consider a property graph with time information in Figure 1.5. As with static graphs, we are interested in the question of node reachability. Such queries are normally supported through the application of a graph pattern with regular path expressions (this is formally defined in Section 2.2.2). It is easy to show that the correctness of the reachability query is compromised. For example, nodes Bill and Alice are shown as connected, but in fact there is no point in time where there is a connected path between them.

Finally, a common approach to model evolving graphs is as a sequence of static graphs. We discuss this approach in detail in Section 6.1, but the gist of our criticism is the issue of expressivity. It is not possible to formulate certain queries over the graph sequence *conceptual model*. Formally, the graph sequence model breaks one of the desired temporal model properties, that of extended snapshot reducibility.

1.2 Property Graph Model and RDF

There is an abundance of various graph models, developed over the past 30 years [8]. Increasingly, two have attained the greatest popularity: the property graph model [2] and the Resource Description Framework (RDF) [59].

The RDF graph is a set of triplets, which can be thought of as edges, since each triplet is a subject, connected to an object with a predicate. The subject and the object are nodes of the graph, while the predicate is the labeled edge. RDF nodes can be literals, resource identifiers, and blank nodes. While an RDF graph can be thought of as a kind of a labeled graph, it is, formally speaking, not, because there is no clear separation between nodes and edges [48]. An edge (triplet) can be identified with a resource descriptor and appear as an object or a subject in another edge. RDF is widely used to encode semantic information on the Web. The RDF standard also includes a graph query language SPARQL, loosely based on SQL.

The property graph model, which we use throughout this dissertation 2.2.1, is in a sense a competing popular graph model. The choice between the two models is largely a matter of philosophy. We selected the property graph model based mainly on the considerations of usability. Specifically:

- In the property model, to add new information about an existing edge the user can simply add new properties. In an RDF graph an edge is a triplet, and new triplets can be added with its descriptor as an object or a subject. However, this is not an easy thing for human graph users to comprehend, because of this lack of separation between nodes and edges. If, on the other hand, we treat an RDF graph as simply a labeled graph, where an edge cannot be a subject/object in another triplet, then adding new information about an existing edge requires modifying a large part of the existing graph [7].

- RDF graphs are more dense than property graphs because even constant literals are encoded as nodes. The property graph model, on the other hand, gives the user the flexibility to encode new information either as new nodes and edges or as properties of existing nodes and edges, whichever makes the most sense for the specific dataset.

- SPARQL has a high learning curve, even though it is somewhat modeled on SQL [19].

- In order to facilitate complex analysis tasks, we wanted a closed algebra, where the result of each query is a TGraph. SPARQL queries can return a graph using the CONSTRUCT clause, but without it return a result set.

On the other hand, RDF is a W3C standard, whereas there is currently no standard for the property graph model and no property graph model query language or algebra.

It is our opinion, therefore, that while neither model wins over the other one, the property graph model is more practical for people, as opposed to the machines. It is finding an increasing support, including by companies such as Oracle [93], in addition to the existing support in the popular Neo4j database system [2]. It is also recognized and accommodated by the Linked Data Benchmark Council[1].

1.3 Motivating Examples

Evolving graph analysis algorithms are numerous, including structural and spatio-temporal graph mining that looks for patterns in space and time and analytics that investigate the evolution of some property of the graph over time.

[1] http://www.ldbcouncil.org/

An interaction network is one typical kind of an evolving graph. It represents people as nodes, and interactions between them such as messages, conversations, and endorsements, as edges. Information describing people and their interactions is represented by node and edge attributes. One easily accessible interaction network is the wiki-talk dataset[2], containing messaging events among Wikipedia contributors over a 13-year period. Information available about the users includes their username, group membership, and the number of Wikipedia edits they made. Messaging events occur when users post on each other's talk pages.

We now present common analysis tasks that motivate our research. We are primarily interested in analyses of the *evolution* of the phenomena the graph represents in the form of queries posed by the user. The examples below were selected for their breadth, based on the analysis of the related work (see Section 6.2).

1.3.1 Node Influence over Time

In an interaction graph, node centrality is a measure of how important or influential nodes are in the graph. Over a dozen different centrality measures exist, providing indicators of how much information "flows" through the node or how the node contributes to the overall cohesiveness of the network. Node importance fluctuates over time. To see whether the wiki-talk graph has high-importance nodes, and how stable node importance is over time during a particular period of interest, we can:

- Look at a subset of the graph that corresponds to the period of interest.
- Compute an importance measure, such as in-degree, for each node and for each point in time.
- Calculate the coefficient of variation per node.

This example demonstrates a need to select a subset of the data corresponding to the period of interest, compute in-/out-degree for each node at each point in time, and compute a single measure across time for each node. Computation of degree is a simple example of a non-temporal *aggregation* operation as defined by the taxonomy of Wood [92]. Aggregation computes a value for each node based on its neighbors and can be used for a wide variety of analyses.

1.3.2 Graph Centrality over Time

Graph centrality is a popular measure that is used to evaluate how connected or centralized the community is. Figure 1.6 visualizes wiki-talk graph centrality evolution over the lifetime of the

[2] `http://dx.doi.org/10.5281/zenodo.49561`

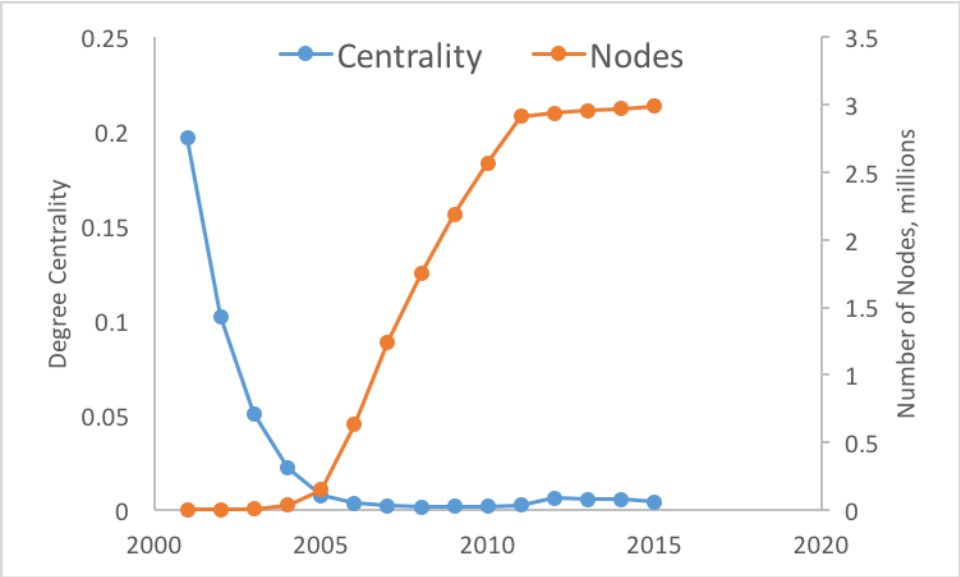


Figure 1.6: Wiki-talk graph centrality over time. The centrality measure is a ratio of the sum of centrality of each node to the maximum possible, a star graph. Wikitalk is a sparse, loosely connected graph. As the size of the graph increases, centrality decreases sharply.

network. This measure can be computed by aggregating in-degree values of graph nodes and may change as communication patterns evolve, or as high influencers appear or disappear. In this case the graph centrality consistently diminishes as the graph grows over time and is large at any point. It is fair to say that the wiki-talk community is quite distributed and not centralized, likely with many different sub-communities that corresponds to our intuition. In sparse interaction graphs there is an additional question of temporal resolution to consider: if two people communicated on May 16, 2010, how long do we consider them to be connected?

This example demonstrates the need to compute graph centrality at every point in its lifetime and to do so at different temporal resolution. For most overall graph centrality measures, the two-step process involves first calculating some measure, such as in-degree or eigenvector centrality, for each graph node, and then accumulating them into one to represent the whole graph. This calls for an operation that can group multiple nodes, in this case all of them, into a new node. Wood terms this operation *node creation* [92] and shows that many graph languages such as GraphQL [49] support it if there is a need to output new nodes that were not part of the input. Creating a single node to represent the whole graph is one way to support computation of some whole-graph measure, but, as we show later, node creation is useful for other types of analyses. Use of temporal windows is also important here to consider different temporal resolution, which is similar to temporal aggregation in temporal relational databases.

1.3.3 Communities over Time

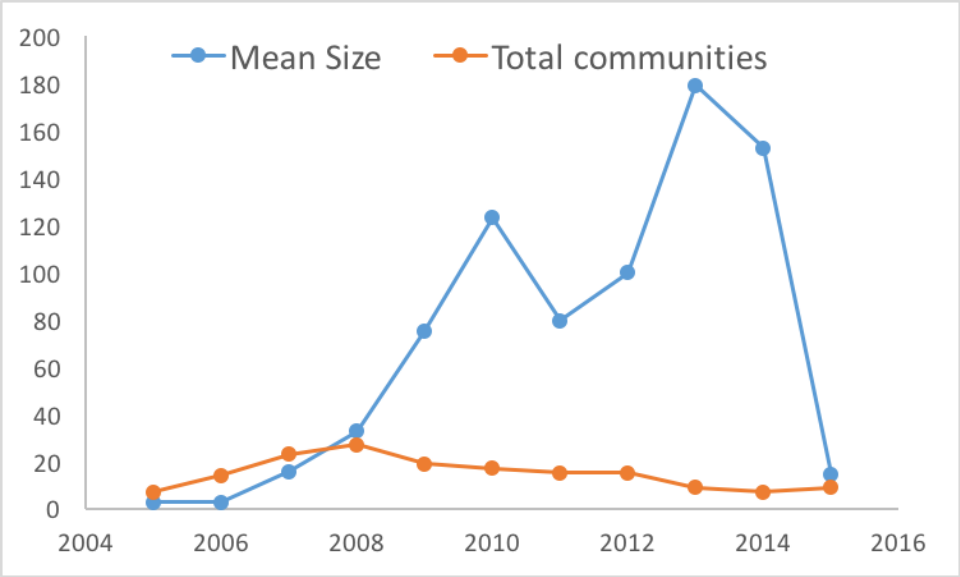


Figure 1.7: Wiki-talk connected components using 1-year temporal resolution.

Interaction networks are sparse because edges are so short-lived. It may be useful to zoom out in time by changing the time granularity of the data such that all durations are, for example, in months or years. This way more items are concurrent and the overall network is more connected. We call this kind of zooming out a change in **temporal resolution** of the graph and we can use it as part of exploratory analysis. After decreasing temporal resolution we can run a community detection algorithm, e.g., compute the connected components of the network, and then consider the number of and size of connected components. Figure 1.7 depicts the evolution of the wiki-talk network communities after zooming out to 1-year resolution. It is evident that communities do form through a pattern of communication, and that such communities can be quite large.

To see whether communities form and at what time scale, we vary the time scale and compute communities, e.g., through connected components detection, group the nodes by the community they form and calculate their size. We can filter out nodes that represent communities below a reasonable threshold, for example of size smaller than two.

This example demonstrates a need to compute graph-wide analytics such as connected components for each point in time, create new nodes that represent some aspect of data of existing nodes, and compute subgraphs.

1.3.4 Spread of Information

We can analyze evolving interaction networks to study how information spreads over time. Suppose each edge has a topic attribute that indicates what each communication between users is about. To see how far and how quickly information spreads, we can compute *journeys*. A journey connects any two nodes by a new edge if there is a path between them such that the edge times are non-decreasing. For example, a journey can be over edges that all co-exist or over edges that follow each other in time. A conditional journey is one where additional predicates on the nodes or edges are specified. In this example, we connect any two nodes that have a path between them on the same topic and assign new edges a length property equal to the sum of the edges in the path. This is an example of an edge creation operation extended for evolving graphs. To see how far information may travel and who the sources are, we can select a subset of the network, consisting only of nodes that originate the longest edges.

1.4 Contributions of this Dissertation

It is the main thesis of this dissertation that extending the property graph model with time in a principled way results in a concise temporal graph algebra that enables usable querying of evolving graphs, overcoming the challenges of previously suggested models.

The contributions of this dissertation are as follows:

- We propose a conceptual representation of an evolving graph, called a TGraph, that captures the evolution of both graph topology and node and edge attributes. This representation builds on both temporal relational models and graph property models.
- To provide systematic support for analysis of evolving graphs, we propose a TGraph algebra, TGA. Our goal in developing TGA is to give users an ability to concisely express a wide range of common analysis tasks over evolving graphs [4]. We present formal properties of TGA, with a focus on its temporal completeness. It is the goal of this thesis to define an algebra that has clear semantics and is sufficiently expressive to support evolving graph analysis for a wide audience of data scientists and researchers.
- We present an implementation in scope of the Portal system, built on Apache Spark / GraphX [44]. This implementation uses non-trivial memory representations and algorithms to support the breadth of TGA with desired performance characteristics.

- We conduct an extensive experimental evaluation with real datasets and compare performance to a published baseline.

1.5 Structure of this Document

The remainder of this dissertation is structured as follows: Chapter 2 provides background information on temporal relational and graph models, including definitions of common algebraic operations. This background is necessary since this work combines the two areas. Chapter 3 defines a conceptual representation of evolving graphs and gives formal definitions of Temporal Graph Algebra (TGA) operators, with detailed examples and a demonstration of how our motivating examples can be supported by the TGA. Chapter 4 describes an implementation of TGA within Apache Spark, including physical representations and optimizations. Chapter 5 provides results of an extensive experimental evaluation of Portal with real datasets in a distributed environment. Chapter 6 reviews relevant published work on evolving graph models, queries, and systems, and places our work in this context. Chapter 7 discusses several directions for future work and our ideas for how it may be accomplished. Finally, Chapter 8 concludes the dissertation with a summary of the contributions and insights.

Chapter 2: Background

A data model consists of a query language, the representation of the objects on which the language operates, an update language, and a mechanism for integrity constraints. This dissertation addresses a representation and an algebra, leaving an update language and integrity constraints for future work. In this chapter we review the foundational work in temporal relational models and in graph models, necessary for understanding the rest of this dissertation.

2.1 Temporal Relational Model

2.1.1 Relations

We assume a linearly ordered, discrete time domain Ω^T where time *instances* have limited precision. In temporal relational databases, a *valid-time* temporal relation schema is represented as $R = (A_1, \ldots, A_m \mid T)$, where $A_1, \ldots, A_m$ are nontemporal attributes with domain Ω_i and T is a temporal attribute over $\Omega^T \times \Omega^T$. The timestamp attribute is special and thus separated by a $\mid$ symbol in the list of attributes. It is also always listed last. This is called *tuple timestamping* [75], since each tuple in a relation is associated with a single time attribute during which it is known to hold. In principle, the time attribute can be a single time instant or a set of instants. We use periods to compactly represent the constituent time points. This is a common representation technique, which does not add expressive power to the data model, compared to associating each tuple with a single time instant [27]. Following the SQL:2011 standard [61], a period (or interval) $t = [s, e)$, a value in the domain T, represents a discrete contiguous set of time instances from domain Ω^T, starting from and including the start time s, continuing to but excluding the end time e.

A tuple $r(a_1, a_2, \ldots, a_n|t)$ over schema R is a finite set over the domain of attributes of R. We refer to the value of a specific attribute A_i in r with notation $r.a_i$. Similarly, the value of the timestamp attribute T for the tuple r is referred to as $r.t$.

Table 2.1: Example temporal relation instance **r**.

	N	S	T
r_1	Susan	30	[2011/5,2015/8)
r_2	Alice	20	[2010/4,2013/4)
r_3	Alice	30	[2013/4,2017/1)

Definition 2.1.1 (Value-equivalent)**.** *Two tuples over the temporal relational schema R are* **value-equivalent** *if they agree on all nontemporal attribute values:*

r *value-equivalent* $r' \iff \forall i\ r.a_i = r'.a_i$

A temporal relation instance **r** over schema R is a finite set of tuples over R. Temporal relations have duplicate-free set-based semantics, i.e., there are no value-equivalent tuples with overlapping time instances.

Example 1. *Consider an instance of the temporal relation in Table 2.1 that is consistent with the described model. Note that there are no value-equivalent tuples with overlapping timestamps. Tuples r_2 and r_3 are consecutive in time but are not value-equivalent because the value of the S attribute is different.*

A *snapshot* of a temporal relation is the state of the relation at one specific instant. Snapshots are obtained with a *timeslice* operation τ_p.

Definition 2.1.2 (Timeslice)**.** *Timeslice operator τ_p maps from a temporal to a nontemporal relation at time point p:*

$\tau_p(\mathbf{r}) = \{(a_1, \ldots, a_m) | (a_1, \ldots, a_m \mid t) \in \mathbf{r} \wedge p \in t\}$

A slice of the example relation instance **r** from Table 2.1 at point 2012/01 contains tuples $(Susan, 30)$ and $(Alice, 20)$.

Definition 2.1.3 (Snapshot-equivalent)**.** *Two temporal relation instances $\mathbf{r}_1$ and $\mathbf{r}_2$ are considered* snapshot equivalent *if their snapshots are equivalent at every time instant [52]:*

$\mathbf{r}_1$ *snapshot-equivalent* $\mathbf{r}_2 \equiv \forall t \in \Omega^T \tau_t(\mathbf{r}_1) \equiv \tau_t(\mathbf{r}_2)$

In this dissertation we use the temporal model with **point semantics**. Point semantics refers to the operations of the temporal algebra rather than the time representation in the model [12]. In other words, the use of intervals for time representation does not dictate specific semantics of the operations.

Point semantics has the following two properties: snapshot reducibility and extended snapshot reducibility.

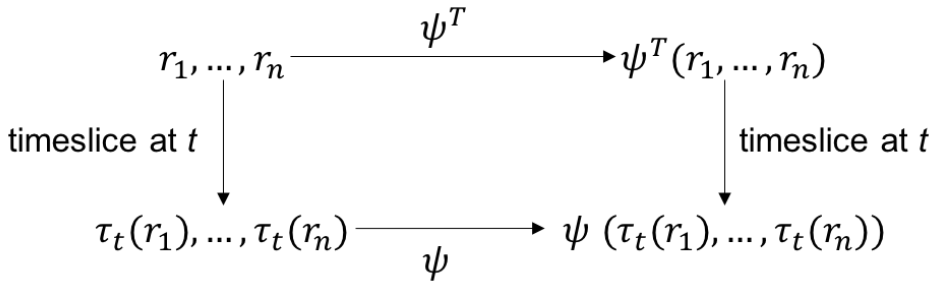


Figure 2.1: Snapshot Reducibility maps each temporal operator to its nontemporal counterpart. Based on [17], Figure 3.

Snapshot reducibility means that a temporal operator applied to a temporal relation produces the same result as some non-temporal operator over corresponding snapshots [16].

Definition 2.1.4 (Snapshot Reducibility)**.** *If ψ^T is an n-ary operator and ψ the corresponding nontemporal operator, then ψ^T is snapshot reducible to ψ iff*

$$\forall t \in \Omega^T(\tau_t(\psi^T(\mathbf{r_1}, \ldots, \mathbf{r_n})) \equiv \psi(\tau_t(\mathbf{r_1}), \ldots, \tau_t(\mathbf{r_n})))$$

Figure 2.1 demonstrates the concept of snapshot reducibility graphically. For instance, if the temporal operator is selection, then it must produce the same result over a temporal relation as a nontemporal selection would produce over each slice of the relation.

Some queries cannot be answered by considering only the snapshots of the temporal relation individually. For example, a user may want to add temporal predicates to limit the results of the temporal selection operator by time or refer to the timestamps of the tuples in a temporal join.

Definition 2.1.5 (Extended Snapshot Reducibility)**.** *Let $k_t(\mathbf{r})$ be a new operator that preserves the timestamp of each tuple on slice, i.e., $k_t(r) = (\tau_t(r), r.T)$, and let ψ^T be an n-ary operator that yields a relation with schema E, with corresponding nontemporal operator ψ. Then ψ^T is extended snapshot reducible to ψ iff*

$$\forall t \in \Omega^T(\tau_t(\psi^T(\mathbf{r_1}, \ldots, \mathbf{r_n})) \equiv \pi_E(\psi(k_t(\mathbf{r_1}), \ldots, k_t(\mathbf{r_n}))))$$

Extended snapshot reducibility allows explicit references to timestamps in the operators by propagating them as data and also requires that the presence of the time references does not change the snapshot reducibility (Def. 2.1.4) of the rest of the statement [16]. Note, however, that the user cannot manipulate the timestamps directly, only reference them in predicates. The k_t operator adds a new nontemporal attribute to a temporal relation, with the value equivalent to the tuple timestamp, and then applies the slice, returning a nontemporal relation. Not every nontemporal operator can have a corresponding temporal operator with extended snapshot reducibility property.

Table 2.2: Uncoalesced instance of relation $\mathbf{r}'$.

N	S	T
Susan	30	[2011/5,2013/5)
Susan	30	[2013/5,2015/8)
Alice	20	[2010/4,2013/4)
Alice	30	[2013/4,2017/1)

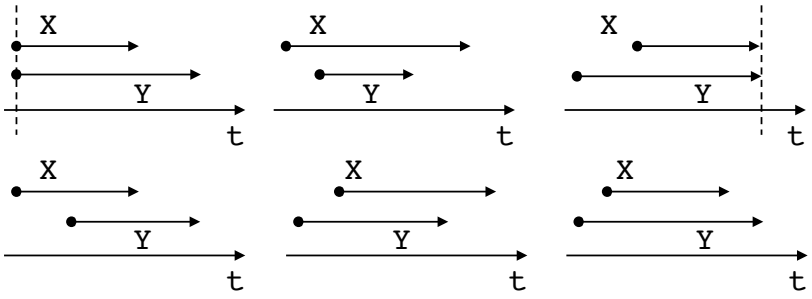


Figure 2.2: X overlaps Y

The set-based binary operators union, intersection, and difference are not schema-robust and thus do not have extended snapshot reducibility [35].

In Section 2.1.3, where we define the operators of the temporal relational algebra, we also give examples with extended snapshot reducibility, e.g., Example 3.

With point semantics, a temporal relation is required to be coalesced – there are no value-equivalent tuples with consecutive timestamps. The statement that $\mathbf{r}$ is coalesced means that each fact is represented once for each time period of maximal length when it holds [13].

Requiring that relations be coalesced is both space-efficient and avoids semantic ambiguity [53]. Consider again relation instance $\mathbf{r}$ from Table 2.1. Compare it to relation instance $\mathbf{r}'$ in Table 2.2. $\mathbf{r}'$ is snapshot-equivalent to $\mathbf{r}$ but not coalesced. If we compute a selection query $\mathtt{q}$ of all tuples lasting longer than 3 years, $q(r)$ will yield tuple $(Susan, 50|[2011/5, 2013/5))$, but $q(r')$ will not.

2.1.2 Temporal Predicates

The SQL:2011 standard [61] defines several predicates over time periods, functionally equivalent to Allen's interval operators [5]. We use these in our algebra. Let X and Y be two time periods, s the start of the period, and e the end of the period.

Definition 2.1.6 (overlaps)**.** $X \, overlaps \, Y \Leftrightarrow X.s < Y.e \wedge Y.s < X.e$

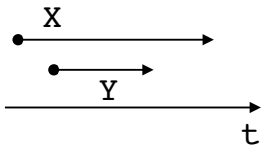


Figure 2.3: X contains Y

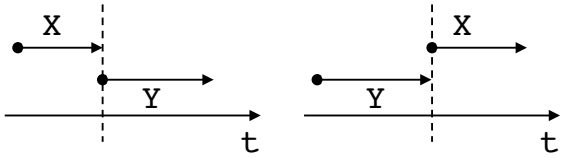


Figure 2.4: X meets Y

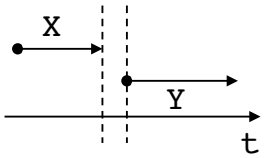


Figure 2.5: X precedes Y

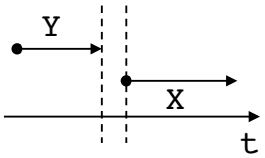


Figure 2.6: X succeeds Y

Informally, two periods are overlapping if they have at least one time instant in common. Figure 2.2 shows all possible cases of two overlapping intervals. This predicate is commutative, i.e., Xoverlaps $Y \Leftrightarrow Y$overlaps X.

Definition 2.1.7 (contains)**.** $X\,contains\ Y \Leftrightarrow X.s <= Y.s \wedge Y.e <= X.e$

Informally, period Y is contained in period X if every instant of Y is in X. Figure 2.3 illustrates containment.

Definition 2.1.8 (meets)**.** $X\,meets\ Y \Leftrightarrow X.s = Y.e \vee Y.s = X.e$

Informally, period X meets period Y if the two periods are immediately one after another. Figure 2.4 shows the two possible cases of meets.

Definition 2.1.9 (precedes)**.** $X\,precedes\ Y \Leftrightarrow X.s < Y.s$

Informally, period X precedes period Y if it is earlier than Y and ends before Y starts, without meeting. Figure 2.5 shows this graphically.

Definition 2.1.10 (precedesOrMeets)**.** $X\,precedesOrMeets\ Y \Leftrightarrow X\,precedes\ Y \vee X.e = Y.s$

Informally, period X precedesOrMeets period Y if X starts and ends before Y.

Definition 2.1.11 (equals)**.** $X\,equals\ Y \Leftrightarrow X.s = Y.s \wedge X.e = Y.e$

Informally, two periods are equal if they share all time instants.

Definition 2.1.12 (succeeds)**.** $X\,succeeds\ Y \Leftrightarrow Y.e < X.s$

Informally, period X succeeds period Y if it starts after Y ends, without meeting. Figure 2.6 demonstrates this graphically.

Table 2.3: Example temporal relation instance **r**.

	N	S	T
r_1	Susan	30	[2011/5,2015/8)
r_2	Alice	20	[2010/4,2013/4)
r_3	Alice	30	[2013/4,2017/1)

Table 2.4: Example temporal relation instance **s**.

	A	B	T
s_1	1	Drexel	[2011/5,2015/8)
s_2	2	UPenn	[2015/10,2017/4)
s_3	3	Temple	[2013/1,2017/1)

Definition 2.1.13 (succeedsOrMeets)**.** $X\,succeedsOrMeets\ Y \Leftrightarrow X\,succeeds\ Y \vee Y.e = X.s$

Informally, period X suceedsOrMeets period Y if X starts at or after Y ends.

Additionally, the following useful operations on periods and dates are supported:

Definition 2.1.14 (datediff)**.** $datediff(X.s, X.e) = X.e - X.s$

datediff returns the difference between two dates. When applied to an interval, it returns its duration.

Definition 2.1.15 (intersect)**.**

$$X\,intersect\ Y = \begin{cases} [max(X.s, Y.s), min(X.e, Y.e)) if\ X\ \ overlaps\ \ Y \\ \emptyset \qquad\qquad\qquad\qquad otherwise \end{cases}$$

intersect returns a period that is the overlap of its two input periods.

Definition 2.1.16 (except)**.**

$$X\,except\ Y = \begin{cases} [X.s, min(X.e, Y.s)) \iff X.s < Y.s \wedge X.e <= Y.e \\ [max(X.s, Y.e), X.e) \iff Y.s <= X.s \wedge Y.e < X.e \end{cases}$$

The except operation returns one or two periods consisting of all time instants that exist in X but not in Y.

2.1.3 Temporal Relational Algebra, TRA

TRA extends relational algebra by specifying how relational operators are applied to temporal relations as defined in Section 2.1.1. Operators are considered temporal if they generate temporal relations when applied to temporal relations and adhere to snapshot reducibility and extended snapshot reducibility as applicable. Let **r** and **s** be temporal relations.

Temporal Selection σ^T

Definition 2.1.17 (Temporal Selection)**.** *Temporal selection* $\sigma^T_\theta(\mathbf{r})$ *where* θ *is a predicate, including temporal predicates defined in Section 2.1.2, is defined as:*

$\sigma^T_\theta(\mathbf{r}) = \{(a_1, \ldots, a_m|t) \mid (a_1, \ldots, a_m|t) \in \mathbf{r} \wedge \theta((a_1, \ldots, a_m|t))\}$

Table 2.5: $\pi_N^T(\mathbf{r})$

N	T
Susan	[2011/5,2015/8)
Alice	[2010/4,2017/1)

Informally, temporal selection returns all tuples in $\mathbf{r}$ that pass predicate θ.

Example 2. *$\sigma^T_{N=Alice}(\mathbf{r})$ computes a temporal selection over example relation instance $\mathbf{r}$ in Table 2.3 to include only tuples with value "Alice" of the N attribute. The result includes tuples r_2 and r_3. The value of the timestamp attribute is not relevant since it is not referenced in θ.*

Example 3. *$\sigma^T_{p\ overlaps\ [2012/1,2013/1)\wedge S>20}(\mathbf{r})$ computes a temporal selection over example relation instance $\mathbf{r}$ from Table 2.3 to include only those tuples that existed at any point in year 2012 with value of the S attribute larger than 20. The result includes only tuples r_1 and r_2, but not r_3, which succeeds the interval in the predicate.*

Temporal Projection π^T

Definition 2.1.18 (Temporal Projection)**.** *Temporal projection $\pi^T_f(\mathbf{r})$, where f is a projection function over nontemporal attributes of R is defined as:*

$$\pi^T_f(\mathbf{r}) = \{(a'_1, \ldots, a_n|t) \mid (a_1, \ldots, a_m|t) \in \mathbf{r} \wedge (a'_1, \ldots, a'_n) = f((a_1, \ldots, a_m))\}$$

This operation may produce value-equivalent tuples with overlapping or consecutive timestamps, so the result must be coalesced.

Example 4. *$\pi^T_N(\mathbf{r})$, where $\mathbf{r}$ is the example relation instance in Table 2.3, produces the relation instance $\mathbf{r}'$ shown in Table 2.5. Observe that two tuples r_2 and r_3 with $N =' Alice'$ in $\mathbf{r}$ are coalesced once the S attribute is projected out. Also note that the timestamp attribute is not referenced directly and cannot be projected out.*

Temporal Aggregation γ^T

Definition 2.1.19 (Temporal Aggregation)**.** *Temporal aggregation ${}_{A_1,\ldots,A_n}\gamma^T_{g_1,\ldots,g_k}(\mathbf{r})$, where $A_1, \ldots, A_n$ is a set of grouping attributes from R and $g_1, \ldots, g_k$ is a set of aggregate functions over remaining attributes of R is defined as:*

Table 2.6: ${}_S\gamma^T_{count(N)}(\mathbf{r})$

S	count(S)	T
20	1	[2010/4,2013/4)
30	1	[2011/5,2013/4)
30	2	[2013/4,2015/8)
40	1	[2015/8,2017/1)

$$
\begin{aligned}
{}_{A_1,\ldots,A_n}\gamma^T_{g_1(A_{n+1}),\ldots,g_k(A_{n+k+1})}(\mathbf{r}) = \{(a_1,\ldots,a_n,s_1,\ldots,s_k|t') \mid \\
(a_1,\ldots,a_n|t) \in \pi^T_{A_1,\ldots,A_n}(\mathbf{r}) \ \wedge \forall x \in \sigma^T_{A_1=a_1,\ldots,A_n=a_n}(\mathbf{r}), x.T \textit{ overlaps } t' \wedge \\
s_1 = g_1(\pi^T_{A_{n+1}}(\sigma^T_{T \textit{ overlaps } t' \wedge A_1=a_1 \wedge \ldots \wedge A_n=a_n}(\mathbf{r}))) \wedge \ldots \ \wedge \\
s_k = g_k(\pi^T_{A_{n+k+1}}(\sigma^T_{T \textit{overlaps } t' \wedge A_1=a_1 \wedge \ldots \wedge A_n=a_n}(\mathbf{r})))\}
\end{aligned}
$$

The second line projects to the grouping attributes from R and states that the timestamp of the result is an intersection of the timestamps of all the tuples that agree on the values of the grouping variables, the third and fourth line compute aggregates over each of the remaining attributes. To make sense of the definition, consider all the tuples that agree on the grouping variables. These form a set over which aggregation functions are applied. A correct result is over the intersection of all the timestamps, i.e., the values of none of the attributes in this group changed during this interval.

Informally, temporal aggregation is equivalent to applying non-temporal aggregation to every snapshot of $\mathbf{r}$ and coalescing the result.

Example 5. ${}_S\gamma^T_{count(N)}(\mathbf{r})$*, where* $\mathbf{r}$ *is the example relation instance in Table 2.3, computes an aggregation over attribute* S *with an aggregate function* count *over the remaining attribute* N*. The timestamp is treated as a special attribute and does not appear either in the grouping attributes or the aggregate functions. The result is depicted in Table 2.6. Observe that the cardinality of the output is higher than of the input. This is because semantically the operation is carried out at each time instant and different groups result at, e.g., 2011/5 with* r_1 *than at 2013/4 with tuples* r_1 *and* r_3*.*

Temporal Cross Product $\times^T$

Definition 2.1.20 (Temporal Cross Product)**.** *Temporal cross product* $\mathbf{r} \times^T \mathbf{s}$*, where* $\circ$ *is a concatenation of two tuples,* $R = (A_1,\ldots,A_m|T)$ *is the schema of* $\mathbf{r}$*,* $S = (B_1,\ldots,B_n|T)$ *is the schema of* $\mathbf{s}$*, is defined as:*

Table 2.7: $\mathbf{r} \times^T \mathbf{s}$

N	S	A	B	T
Susan	50	1	Drexel	[2011/5,2015/8)
Susan	50	3	Temple	[2013/1,2015/8)
Alice	20	1	Drexel	[2011/5,2013/4)
Alice	20	3	Temple	[2013/1,2013/4)
Alice	30	1	Drexel	[2013/4,2015/8)
Alice	30	2	UPenn	[2015/10,2017/1)
Alice	30	3	Temple	[2013/4,2017/1)

$\mathbf{r} \times^T \mathbf{s} = \{(a_1, \ldots, a_m, b_1, \ldots, b_n | t_1 \textit{ intersect } t_2) \mid (a_1, \ldots, a_m | t_1) \in \mathbf{r} \wedge (b_1, \ldots, b_n | t_2) \in \mathbf{s} \wedge t_1 \textit{ overlaps } t_2\}$

Informally, temporal cross product applies non-temporal cross product to every snapshot pair of **r** and **s** and the result is coalesced.

Example 6. $\mathbf{r} \times^T \mathbf{s}$*, where* **r** *and* **s** *are the example relation instances in Tables 2.3 and 2.4 respectively, computes a result shown in Table 2.7. Similarly to aggregation, the cardinality is different than would be expected in the nontemporal variant due to different groups formed at different time instants. For example,* $r_1 \circ s_2$ *is not produced because there are no time instances when both tuples hold.*

Temporal Join $\bowtie^T$

Definition 2.1.21 (Temporal Join)**.** *Temporal join* $\mathbf{r} \bowtie_\theta^T \mathbf{s}$*, where* θ *is a join condition, is defined as:*

$\mathbf{r} \bowtie_\theta^T \mathbf{s} = \sigma_\theta^T(\mathbf{r} \times^T \mathbf{s})$

Just as a regular relational join is defined through selection applied to the result of a cross product, so is the temporal join. Temporal outer and semi joins are defined similarly.

Temporal Union $\cup^T$

Definition 2.1.22 (Temporal Union)**.** *Temporal union* $\mathbf{r} \cup^T \mathbf{s}$*, where* **r** *and* **s** *are union-compatible relations with schema R, is defined as:*

$\mathbf{r} \cup^T \mathbf{s} = \{(a_1, \ldots, a_m | t) \mid (a_1, \ldots, a_m | t) \in \mathbf{r} \vee (a_1, \ldots, a_m | t) \in \mathbf{s}\}$

The result of a temporal union requires coalescing as it is possible that two value-equivalent tuples exist in consecutive or overlapping time periods.

Table 2.8: Example temporal relation instance **v**.

	N	S	T
v_1	Susan	30	[2012/5,2013/6)
v_2	Alice	20	[2012/1,2015/1)

Table 2.9: $\mathbf{r} \cup^T \mathbf{v}$

N	S	T
Susan	30	[2011/5,2015/8)
Alice	20	[2010/4,2015/1)
Alice	30	[2013/4,2017/1)

Table 2.10: $\mathbf{r} \cap^T \mathbf{v}$

N	S	T
Susan	30	[2012/5,2013/6)
Alice	20	[2012/1,2013/4)

Table 2.11: $\mathbf{r} \setminus^T \mathbf{v}$

N	S	T
Susan	30	[2011/5,2012/5)
Susan	30	[2013/6,2015/8)
Alice	20	[2010/4,2012/1)
Alice	30	[2013/4,2017/1)

Example 7. *Consider example relation instance* **v** *that is union-compatible with* **r**. $\mathbf{r} \cup^T \mathbf{v}$ *computes a temporal union of* **r** *and* **v**, *the result of which is depicted in Table 2.9. The first tuple is produced from tuples* r_1 *and* v_1 *and its timestamp is the union of the two periods. Also observe that two tuples with* $N =' Alice'$ *are produced over overlapping time periods, e.g. between 2013/4 and 2015/1. Since the two tuples are not value-equivalent, they are not coalesced into one.*

Temporal Intersection $\cap^T$

Definition 2.1.23 (Temporal Intersection)**.** *Temporal intersection* $\mathbf{r} \cap^T \mathbf{s}$, *where* **r** *and* **s** *are union-compatible relations with schema R, is defined as:*

$$\mathbf{r} \cap^T \mathbf{s} = \{(a_1, \ldots, a_m | t_1 \textit{ intersect } t_2) \mid (a_1, \ldots, a_m | t_1) \in \mathbf{r} \wedge (a_1, \ldots, a_m | t_2) \in \mathbf{s} \wedge t_1 \textit{ overlaps } t_2\}$$

Example 8. *Consider again union-compatible example relation instances* **r** *and* **v**. $\mathbf{r} \cap^T \mathbf{v}$ *computes a temporal intersection, the result of which is depicted in Table 2.10. Both tuples in the result are produced only for the time periods when they existed in* both *relations.*

Temporal Difference $\setminus^T$

Definition 2.1.24 (Temporal Difference)**.** *Temporal difference* $\mathbf{r} \setminus^T \mathbf{s}$, *where* **r** *and* **s** *are union-compatible relations with schema R, is defined as:*

$$\mathbf{r} \setminus^T \mathbf{s} = \{(a_1, \ldots, a_m | t_1) \mid (a_1, \ldots, a_m | t_1) \in \mathbf{r} \wedge \nexists (a_1, \ldots, a_m | t_2) \in \mathbf{s} \wedge t_1 overlaps t_2\} \cup$$

$$\{(a_1, \ldots, a_m | t_1 \textit{ except } t_2) \mid (a_1, \ldots, a_m | t_1) \in \mathbf{r} \wedge \exists (a_1, \ldots, a_m | t_2) \in \mathbf{s} \wedge t_1 overlaps t_2\}$$

Table 2.12: Example nested temporal relation instance **n**.

A	C		T
Susan	N	G	[2016/9,2016/12)
	CS101	A	
	INFO210	A-	
	ECON101	B	
Alice	N	G	[2016/9,2016/12)
	CS101	F	
Alice	N	G	[2017/1,2017/4)

Example 9. *Consider again union-compatible example relation instances* $\mathbf{r}$ *and* $\mathbf{v}$. $\mathbf{r} \setminus^T \mathbf{v}$ *computes a temporal difference, the result of which is depicted in Table 2.11. Observe that* $r_1 \setminus v_1$ *produced two tuples because* $r_1.t$ *contains* $v_1.t$.

2.1.4 Nested Temporal Relational Algebra

In this work we use *nested* temporal relations, and, more specifically, V-relations [3].

A valid-time nested temporal relation schema is represented as $R = (A_1, \ldots, A_m \mid T)$, where $A_1, \ldots, A_m$ are nontemporal attributes with domain $\Omega_i | \{S\}$, where S is a nontemporal V-relation. A V-relation is a complex value relation where:

- Set and tuple constructors are required to alternate. Another way to state this is that at any level the tuple schema must consist of only atomic types and sets of tuples, but at least one of them must be atomic.
- For each set of tuples, the atomic attributes form a key.

Example 10. *Consider a nested temporal relation in Table 2.12, containing class registration and grade information. Its schema is* $(A, C : \{(N, G)\} | T)$, *where* C *is a nested attribute. Note that there are two records for* Alice, *in different time periods. V-relations must have at least one atomic attribute on each level of nesting. In this example attribute* A *forms the key on the top level, and attributes* N *and* G *form a composite key in the nested relation of attribute* C. *Tuple with key* Alice *has an empty set in the* C *attribute during period* $[2017/1, 2017/4)$.

To convert between nested and unnested relations, nest and unnest operations are required.

Nest ν^T

Definition 2.1.25 (Nest)**.** *Nest* $\nu^T_{b,a_1,\ldots,a_n}(\mathbf{r})$ *where* $\mathbf{r}$ *is a temporal relation,* $a_1, \ldots, a_n$ *is a subset of attributes over schema of* $\mathbf{r}$, $s_1, \ldots, s_k$ *is a set of remaining attributes of* $\mathbf{r}$, *and* b *is the new attribute name, is defined as:*

Table 2.13: $\mu^T_C(\mathbf{n})$

A	N	G	T
Susan	CS101	A	[2016/9,2016/12)
Susan	INFO210	B	[2016/9,2016/12)
Susan	ECON101	B	[2016/9,2016/12)
Alice	CS101	F	[2016/9,2016/12)

$$\nu^T_{b,a_1,\ldots,a_n}(\mathbf{r}) = \{(b, s_1, \ldots, s_k|t) \mid (a_1, \ldots, a_n, s_1, \ldots, s_k|t) \in \mathbf{r} \wedge b = (a_1, \ldots, a_n)\}$$

Unnest μ^T

Definition 2.1.26 (Unnest)**.** *Unnest $\mu^T_b(\mathbf{r})$ where $\mathbf{r}$ is a nested temporal relation and b is a nested attribute of $\mathbf{r}$ consisting of attributes $a_1, \ldots, a_n$, is defined as:*

$$\mu^T_b(\mathbf{r}) = \{(a_1, \ldots, a_n, s_1, \ldots, s_k|t) \mid (b, s_1, \ldots, s_k|t) \in \mathbf{r} \wedge b = (a_1, \ldots, a_n)\}$$

Unnest is an inverse of nest, but not necessarily vice-versa [3]. One reason this is so is because unnesting an empty set attribute for a tuple results in removal of that tuple from the result.

Example 11. *Consider the example nested relation instance* $\mathbf{n}$ *in Table 2.12. The result of applying the unnest operator on attribute C results in a new relation instance, depicted in Table 2.13. While the nested relation* $\mathbf{n}$ *had 3 tuples, the result of unnesting has 4, since the tuple with key* Susan *has a set of 3 tuples in its nested attribute C, while the second tuple for* Alice *is removed as a result of unnesting an empty set.*

2.2 Graph Model

We have discussed TRA – a temporal algebra for relations. We now shift our attention to non-temporal graph algebras.

2.2.1 Property Graph Model

Definition 2.2.1 (Graph)**.** *A graph G is 6-tuple $(V, E, \Pi, \rho, \lambda_v, \lambda_e)$, where:*

- *V is a finite set of* nodes*,*
- *E is a finite set of* edges*,*
- *Π is a set of available properties, where each element in Π is a pair (L_i, D_i), L_i is a property name, and D_i is the domain of property L_i,*
- *$\rho : E \to (V \times V)$ is a total function that maps an edge to its source and destination node,*

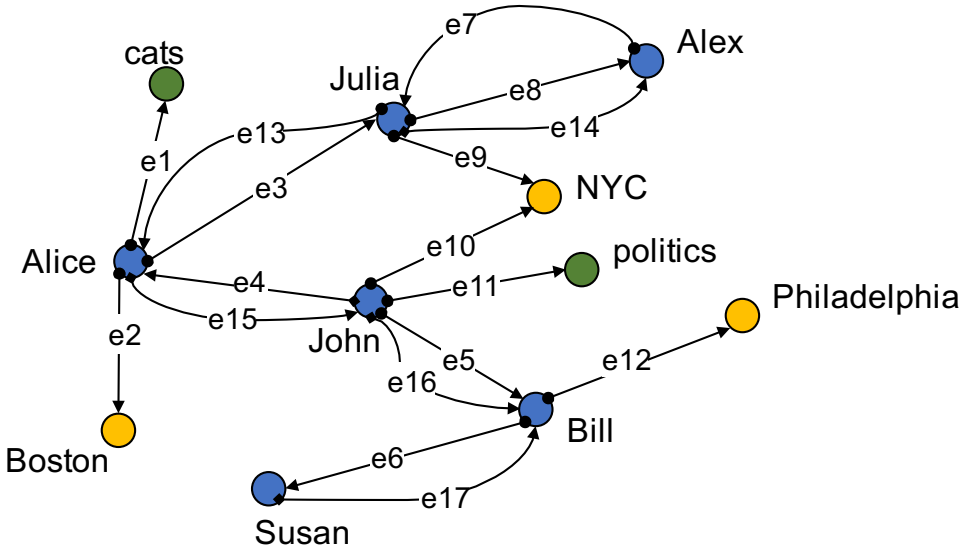


Figure 2.7: Example graph G of a social network.

Table 2.14: Properties of graph G.

Entity	**Property Set**
Boston	type=city
Philadelphia	type=city
NYC	type=city, altname=New York City
e1	type=likes
e2	type=livesIn
e3	type=friendOf
...	
e7	type=follows
e8	type=friendOf

- $\lambda_v : V \times \bigcup L_i \rightarrow Val$ *is a partial function that maps a node from* V *and a property label in* L *to a value in domain* D*,*
- $\lambda_e : E \times \bigcup L_i \rightarrow Val$ *is a partial function that maps an edge in* E *and a property label in* L *to a value of that property in its domain* D*.*

This definition is taken from a survey by Angles et al. [7], with the node and edge labels removed. A node, resp. edge, label is expressed as a value of a property label type and is thus subsumed.

The property set of each node or edge may **not** be empty, since the property with label type is required, and D_i is not restricted to be of atomic types, and may, e.g., be sets, maps, or tuples. For convenience, we can refer to the instance of set of nodes V in an instance of a graph G, we use the notation $G.V$, and similarly for E, etc.

G is a *multigraph* since multiple edges can exist between any pair of nodes. For example, in a social network two people may be connected by one edge indicating a friend status and another

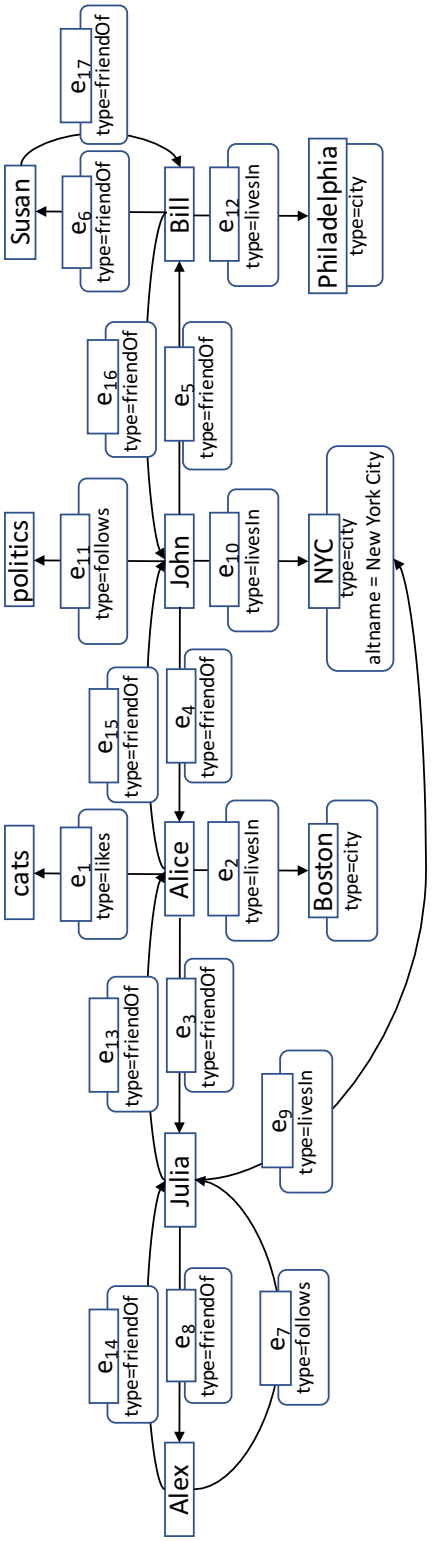


Figure 2.8: Example graph G depicted with property sets. Type of all people nodes, i.e., Julia, John, etc., is not shown for readability.

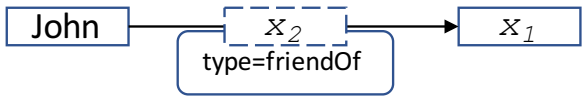


Figure 2.9: Example basic pattern P_1.

edge to indicate a follows relationship. Figure 2.7 shows an example graph of a social network and Table 2.14 its attributes. Each node has a unique id, e.g., $Alice, Philadelphia, \ldots$, as well as each edge has a unique id, e.g., $e1, e2, \ldots$. All nodes and edges have properties. Observe, for example, that node NYC has a property type with value city and another property altname with value New York City. There are three edges, $e7$, $e8$, and $e14$ between nodes Julia and Alex. Figure 2.8 shows the same graph but with property sets included.

One way to represent graph G is with a pair of nested relations V and E, where V has a schema $(\underline{ID}, ATTR)$ and E has a schema $(\underline{ID}, SRCID, DSTID, ATTR)$, and there is a foreign key constraint from E to V.

2.2.2 Graph Queries

There is no single standard graph algebra. This section reviews the most common graph operations, based on the survey by Wood [92], but adapted for the property model, and a survey by Angles et al. [7]. It provides some background for the proposed temporal graph algebra in Chapter 3 and is not meant to be a comprehensive survey of graph algebras.

Subgraph Matching

The most common graph operation is subgraph matching, which finds all subgraphs within the input graph matching a pattern. Subgraph matching exists in several forms, from a basic graph pattern to an extended graph pattern. Let G be a graph and $x, y, z, \ldots$ be a countable set of variables.

Definition 2.2.2 (Basic Graph Pattern)**.** *A basic graph pattern (BGP) is a graph P, where V is extended with a set of node variables, E is extended with a set of edge variables, and property names and values are also extended with variables.*

Essentially a basic graph pattern is a graph where nodes, edges, property names, and property values may be variables. For readability, we will express patterns visually in this work, although they can be expressed as conjunctive queries or enumerations of each set and mapping of P.

Table 2.15: Set of bindings of variables of P_1 in G.

x_1	x_2
Alice	$\mathbf{e_4}$
Bill	$\mathbf{e_5}$

Example 12. *Assume we want to find all friends of John for the example social graph in Figure 2.8. We can formulate a basic pattern depicted in Figure 2.9. The edge expressions are in dashed boxes for readability. This pattern corresponds to the following graph tuple:*

$V = \{\mathit{John}, x_1\}$

$E = \{x_2\}$

$\Pi = \{(type, String)\}$

$\rho(x_2) = (\mathit{John}, x_1)$

$\lambda_e(x_2, type) = \mathit{friendOf}$

The pattern itself is a graph, as defined in Def. 2.2.2. To use a basic pattern as input in a subgraph query, we need to define the concept of a *match*.

Definition 2.2.3 (Match)**.** *Given a graph G and a basic graph pattern P, a match h of P in G is a mapping from constants and variables in P to constants in G such that:*

- *for each constant a, $h(a) = a$,*
- *for each node variable x in P, it holds that $h(x) \in G.V$,*
- *for each edge variable y in P, it holds that $h(y) \in G.E$,*
- *for each property variable z in P, it holds that $h(z) \in G.\Pi.L$,*
- *for each property variable a in Val, it holds that $h(a) \in G.Val$,*
- *the structure of P is preserved in its image under h in G.*

In other words, when h is applied to all the terms of P, the result is a sub-graph of G. An evaluation of a graph pattern P against a graph G corresponds to the set of all possible matches of P with respect to G.

Two semantic types are possible with respect to whether multiple variables in P can map to the same term in G: a *homomorphism-based* semantics (no constraints) and a *isomorphism-based* semantics (some constraints on what can be repeated) [7]. The most common semantics is the unconstrained one and it is used, for example, by the SPARQL query language [59].

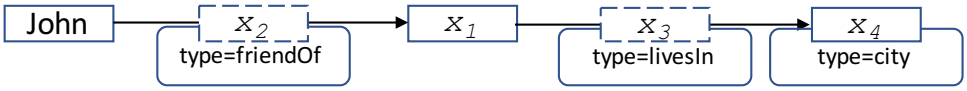


Figure 2.10: Example basic pattern P_2.

Table 2.16: Set of bindings of variables of P_2 in G.

x_1	x_2	x_3	x_4
Alice	$\mathbf{e_4}$	$\mathbf{e_2}$	Boston
Bill	$\mathbf{e_5}$	$\mathbf{e_{12}}$	Philadelphia

Example 13. *Consider the example graph G in Figure 2.8 and the graph pattern in Figure 2.9. The set of bindings of variables of P_1 in G is shown in Table 2.15. John only has two friends, Alice and Bill.*

Example 14. *Consider again the example graph G in Figure 2.8 and a different graph pattern P_2 in Figure 2.10. The set of bindings of variables of P_2 in G is shown in Table 2.16. The pattern models not only friends of John but also cities where they live.*

An evaluation of a graph pattern over a graph returns a set of matches, i.e., bindings for variables in P. It can also be defined to be a Boolean query returning only whether a graph matches the pattern, i.e., whether the set of matches is not empty, or individual subgraphs matching the pattern, i.e., separate graphs, one for each match. Other variants exist, but for the purposes of this work we are interested in a closed graph algebra. Thus we want to define a subgraph operation that takes in a graph pattern and a graph and returns another graph.

Definition 2.2.4 (Subgraph)**.** *A subgraph operation subgraph(G,P), where G is a graph from Definition 2.2.1 and P is a graph pattern from Definition 2.2.2 is defined as:*

$\mathit{subgraph}(P, G) = (V', E', \Pi, \rho, \lambda_v, \lambda_e)$,

where $V' = q_v(P, G)$ is a set of all node matches (per Def. 2.2.3) of node variables and constants in P and $E' = q_e(P, G)$ is a set of all edge matches of edge variables and constants in P.

Informally, the result is a subgraph of input graph G consisting of all matches h of P in G. This query type can be also expressed as a relational conjunctive query over the graph node and edge relations.

Example 15. *Consider an example graph in Figure 2.8 and pattern P_1 in Figure 2.9. subgraph(G,P_1) returns a new graph depicted in Figure 2.11, consisting of node John and only of people who are friends of John.*

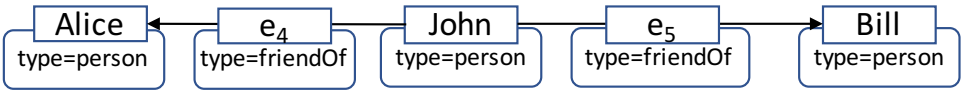


Figure 2.11: subgraph(G,P_1)

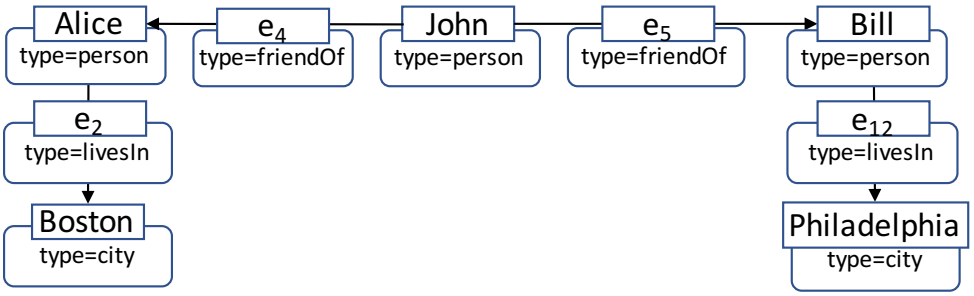


Figure 2.12: subgraph(G,P_2)

Example 16. *Consider again the example graph in Figure 2.8 and another pattern P_2 in Figure 2.10.* subgraph*(G,P_2) returns not only John's friends but also places where they live, as shown in Figure 2.12.*

This form of subgraph query, accepting a basic graph pattern, is the most basic form of subgraph matching. Some systems only support basic pattern subgraph queries. For instance, Spark GraphX [44] includes subgraph operation, which accepts a node and edge triplet predicates expressed as boolean lambda functions. SocialScope [6] includes a Node Selection and Link Selection operators that are essentially conjunctive queries over nodes and edges respectively. Additionally, it is useful to be able to express *path* queries of any length over graphs. For this we define a *navigational graph pattern* [7]. But first we need to formally state what a *path* is.

Definition 2.2.5 (Path)**.** *A path p between nodes v_0 and v_m in G is a sequence $v_0e_0v_1e_1v_2 \ldots v_{m-1}e_{m-1}v_m$, where $m \geq 0, v_i \in V(1 \leq i \leq m), e_i \in E(1 \leq i < m)$, and $p_e(e_i) = (v_i, v_{i+1})(1 \leq i < m)$. The length of p is m.*

An *empty path* is a path of length 0, i.e., just a single node and no edge.

Definition 2.2.6 (Regular Path Expression)**.** *A regular path expression (RPE) over graph G is an expression of the form*

$RPE = x \xrightarrow{\alpha} y$,

where α specifies a regular expression over the edges and their properties, denoting a path between x and y. $|$ signifies a disjunction, $\cdot$ a concatenation, $+$ one or more occurrences, ? for zero or one occurrence, .. for a numerical range, and $*$ for no constraints.

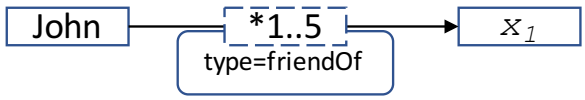


Figure 2.13: Example navigational graph pattern P_3.

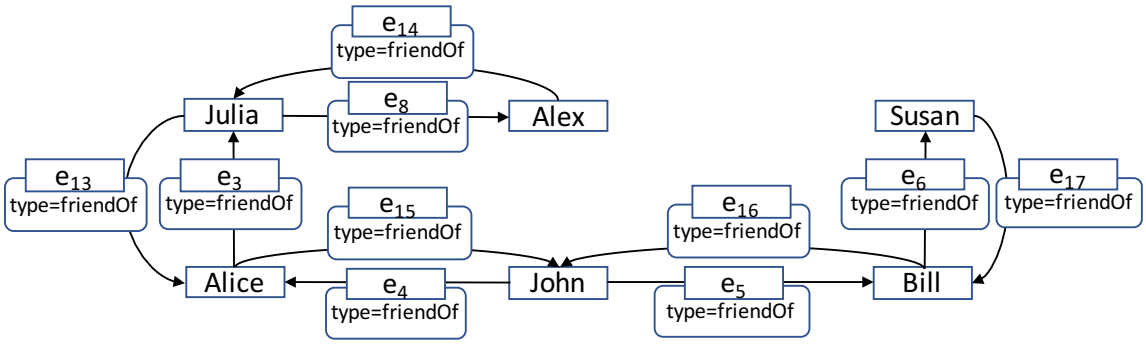


Figure 2.14: subgraph(G,P_3). Properties of the people nodes not shown for readability.

Definition 2.2.7 (Navigational Graph Pattern)**.** *A navigational graph pattern (NGP) is a graph NGP, where V is extended with a set of node variables, E is extended with a set of edge variables and regular path expressions (RPEs), and property names and values are also extended with variables.*

Informally, NGP is similar to BGP, but the nodes are connected by paths specified by a regular expression. We use the arbitrary path semantics, i.e., a regular path expression has a match in the input graph if there is any path matching the RPE, without the need to enumerate all the path matches. If the path expression contains a property constant, then every edge in the matching path must contain that property name with the value specified by the constant.

Most full-fledged graph databases such as the popular Neo4j [2] support NGPs, although the exact pattern language varies. We can now extend the definition of subgraph to accept NGPs in addition to BGPs.

Example 17. *Consider again the example graph G in Figure 2.8. We can select everyone who is connected to John by a friendship path of no longer than 5 hops using a navigational graph pattern depicted in Figure 2.13. The result of a* subgraph(G, P_3) *is shown in Figure 2.14.*

Navigational graph patterns can express a wide range of queries, including neighborhood of a specific node, connected components, and many others.

Figure 2.15: Example navigation graph pattern P_4.

Example 18. *To compute a subgraph of our running example graph G equal to a neighborhood of* John*, we can use the NGP P_4, depicted in Figure 2.15. As it happens, the result is G itself as every node is connected to* John *by some path.*

The complexity of the subgraph operation depends on the specifics of the pattern, but is in general NP-complete [7]. Under data complexity, however, it can be solved in polynomial time [3]. Furthermore, even under combined complexity there are many identified cases where an efficient solution is possible.

Aggregation

It is useful to be able to compute new properties of nodes based on their neighbors, such as a node degree or a list of all cities that one's friends have visited. An aggregation operation achieves this purpose.

Definition 2.2.8 (Aggregation)**.** *Graph aggregation* agg(G, P^{agg})*, where P is a navigational graph pattern extended with aggregation functions and M is a set of aggregating properties in P is defined as:*

agg$(G, P^{agg}) = (V, E, \Pi', \rho, \lambda'_v, \lambda'_e)$*, where*

- $\Pi' = \Pi \cup M$,
- $Dom(\lambda'_v) = Dom(\lambda_v) \cup A_v$,
- $Dom(\lambda'_e) = Dom(\lambda_e) \cup A_e$,
- A_v *is the set of values of all node aggregating expressions in P applied to G,*
- A_e *is the set of values of all edge aggregating expressions in P applied to G.*

The set M of aggregating properties in P are the property names in P, the value of which is specified by the recognized aggregation functions. The aggregating expressions are over the node and edge constants and variables in P, using arbitrary path semantics.

Informally, graph aggregation produces a new graph G' that is isomorphic to G, where each node matching a node in P with an aggregation statement has a new property with the value of that aggregation. The aggregation is over other variables of P, sometimes referred to as *collecting variables*. The usual relational aggregation functions sum, min, max, and count are supported. Additionally, set-based aggregation functions may be included to, e.g., place all the group values into a set.

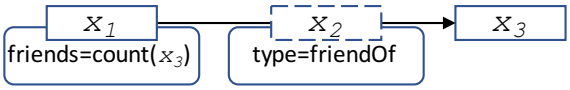


Figure 2.16: Example navigational graph pattern with aggregation, P_5.

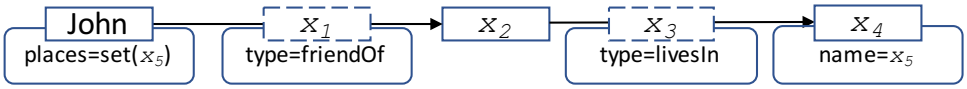


Figure 2.17: Example navigational graph pattern with aggregation, P_6.

Aggregation queries can be translated into recursive Datalog programs with aggregation. To ensure termination in such cases only monotonic aggregate functions are used, as shown in [32]. For this reason, aggregation function average is usually not supported.

Example 19. *Consider again the example social graph in Figure 2.8.* agg(G, P_5)*, where* P_5 *is an* NGP^{agg} *depicted in Figure 2.16, produces a new property named* friends *for each node that has at least one outgoing* friendOf *edge equal to the number of such edges. The property value is obtained using the* count *aggregation function over the collecting variable* x_3*. Thus node* Alice *has a new property* friends *with value 2, as do nodes* John*,* Bill*, and* Julia*. Nodes* Susan *and* Alex *also have this new property, but with value 1.*

Example 20. *To compute a set of all cities where friends of node* John *live, we can use an* NGP^{agg} *depicted in Figure 2.17. Assume that every place node in* G *has a property name. Notice that this pattern is almost identical to pattern* P_2 *from Figure 2.10, with the addition of the aggregating property* places *over collecting variable* x_5*. The result of* agg(G, P_6) *is graph* G*, but with a new property* places *for node* John*, equal to a set {Philadelphia,Boston,NYC}.*

Example 21. *Since aggregation supports recursion, complex whole-graph computation can be expressed. Consider again the graph in our running example, but assume that each edge has a property* distance *that is equal to 1. We can compute the length of the shortest friendship path to target node* John *with the* NGP^{agg} *depicted in Figure 2.18.* distance *is the collecting variable,* sp *is the aggregating property, and the aggregation expression takes the sum over the distances of each path between nodes* x_1 *and* John *and picks the smallest.*

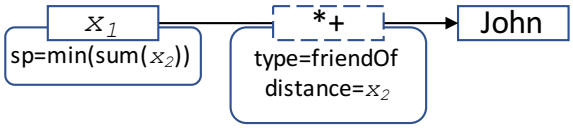


Figure 2.18: Example navigational graph pattern with aggregation, P_7.

Graph aggregation can be performed in PTIME when the aggregation functions form a closed semiring [92]. For example, the length of a shortest path can be computed in time $O(|G|^2 \cdot |L|)$, where L is the regular expression [7]. However, in the general case the problem may not terminate.

Graph aggregation is supported in Spark GraphX by a nonrecursive aggregateMessages method that accepts an edge triplet predicate function and a commutative associative aggregation function. SocialScope defines a Node Aggregation function that is similar but accepts a wider range of aggregation functions. Since this approach does not allow for recursion, separate analytics methods are added typically using a Bulk Synchronous Parallel execution model [89] such as Pregel [71]. We will return to this point in Chapter 4.

Map

While not as common, a map operation on graph nodes or edges allows to modify node (resp. edge) properties by applying an arbitrary mapping function, similar to relational projection. This operation is present in, e.g., Spark GraphX [44] and Neo4j [2].

Definition 2.2.9 (Vertex-Map)**.** *The vertex-map operator, denoted* $\mathsf{map}_v(f_v, G)$*, where* f_v *is a user-defined function, is defined as:*

$\mathsf{map}_v(f_v, G) = (V, E, \Pi, \rho, \lambda'_v, \lambda_e)$,

where $\lambda'_v(v, l) = f_v(v, \{(l_i, d_i) \mid l_i \in L_i \wedge d_i = \lambda_v(v, l_i)\}).l$.

The f_v *function takes a single node, with all its properties, as input, and outputs a modified set of node properties associated with a set of valid domain values. We require that* f_v *can be executed in PTIME in relation to the number of node properties.*

The edge-map operator is defined similarly, with the f_e function applied to each edge tuple $(e, v1, v2, a)$.

Both vertex- and edge-map operations have linear data complexity.

Node Creation

So far we have maintained a closed world semantics, i.e., for each graph G' being in the output of a query, $G'.V \subseteq G.V, G'.E \subseteq G.E$. However, it is sometimes useful to be able to output new nodes that are not part of the input. The operation that allows the addition of new nodes is called **node creation** [92].

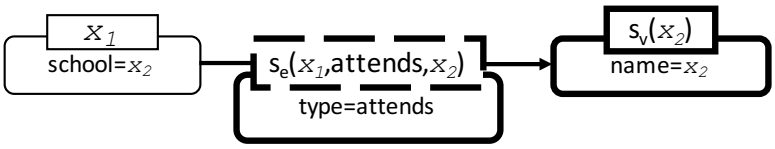


Figure 2.19: Example NGP with Skolemization, P_8.

Recollect that nodes in G have identity. In order to create new nodes, a mechanism to assign new identifiers is required. One way to accomplish that is using a Skolem function, a concept originally used in predicate logic.

Definition 2.2.10 (Node creation)**.** *A node creation operator* $\mathsf{node}(G, P^s)$*, where* G *is a graph, and* P^s *is a navigational graph pattern extended with Skolem functions, is defined as:*

$\mathsf{node}(G, P^s) = (V \cup M, E \cup N, \Pi, \rho', \lambda'_v, \lambda'_e)$*, where*

- M *is a set of new nodes created by the Skolem function* s_v *based on the matches of the variables used as parameters to* s_v *in* P^s*,*
- N *is a set of edges created by the Skolem function* s_e *based on the matches of the variables used as parameters to* s_e *in* P^s*,*
- ρ' *is extended to include mappings for new edges in* N*,*
- λ'_v *is extended to include properties for new nodes in* M*,*
- λ'_e *is extended to include properties for new edges in* N*.*

Informally, node creation returns graph G with additional nodes representing matches of P^s in G and identifiers assigned by Skolem functions, and new edges connecting the new nodes to elements of the matches, as depicted by P^s.

Example 22. *Consider the graph* G *in our running example. Assume that some nodes have a property* school*. We can create new nodes to represent each school that people attend using the pattern* P_8*, depicted in Figure 2.19. New nodes and edges are depicted with a bold outline for readability, but in principle are identified by the use of the Skolem functions. Note the use of a Skolem function* s_v *to assign identity to new school nodes and another Skolem function* s_e *to assign identity to new* attends *edges connecting to new nodes. Assuming that both* Alice *and* John *have a property* school *with value* Drexel*, we add a new node* Drexel*, as depicted in Figure 2.20.*

Example 23. *For a more complex case, consider pattern* P_9*, shown in Figure 2.21. This pattern creates a new node to represent all friends (friends-of-friends, etc.) of* John *who share his interests. Every node with a friendship path from John and an edge indicating an interest that is shared by*

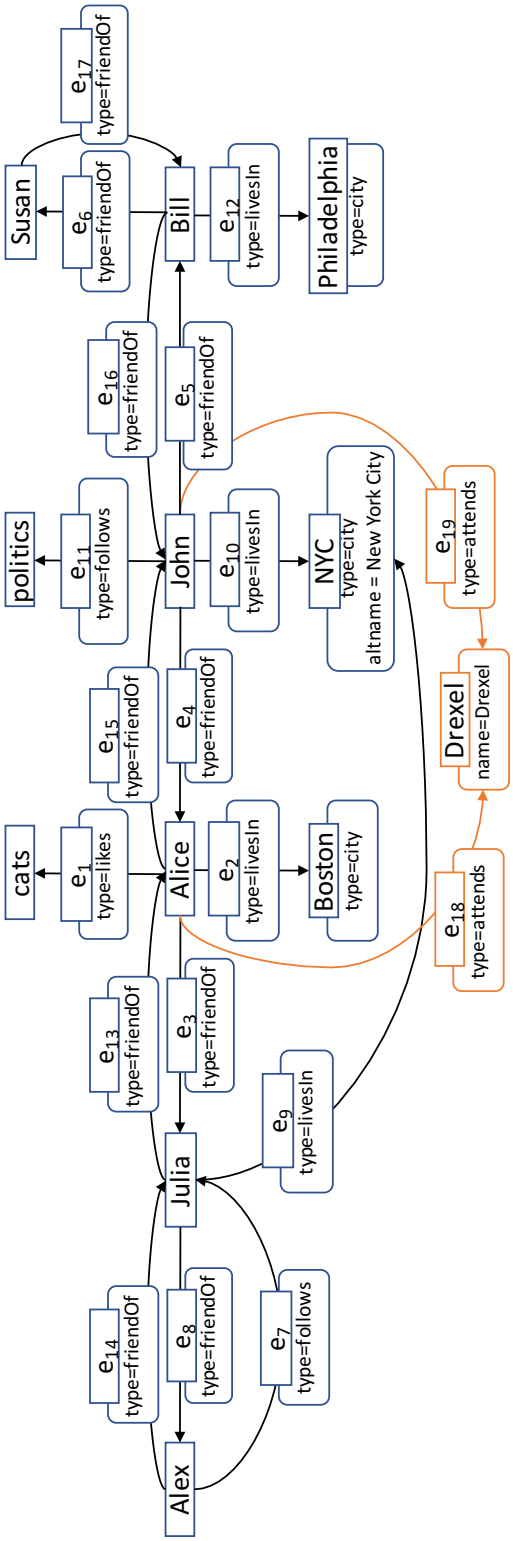


Figure 2.20: node(G, P_8). New nodes and edges are highlighted in orange. Properties of the people nodes are not shown for readability.

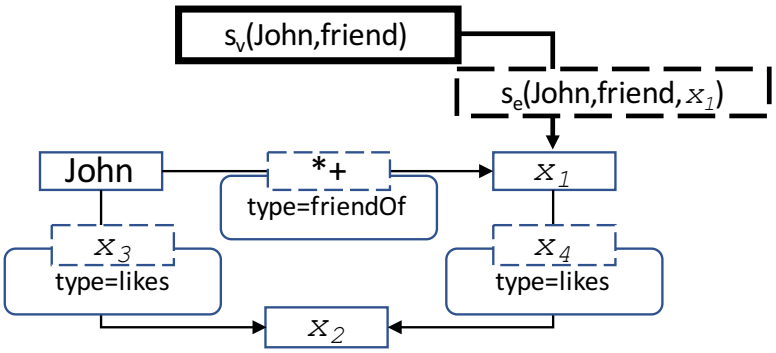


Figure 2.21: Example NGP with Skolemization, P_9.

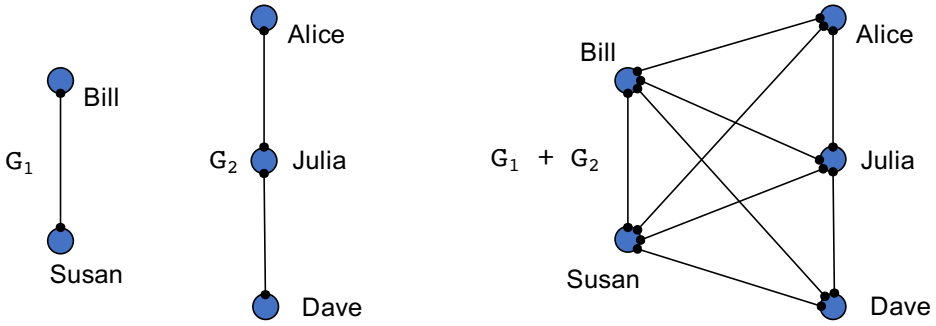


Figure 2.22: Example join of two graphs G_1 and G_2.

John is connected to the new node. As it happens, there are no matches in G for this pattern, so no new nodes are created and the result is G.

Node creation involves pattern matching, followed by creating nodes and edges associated with the matches. The combined complexity of pattern matching is known to be NP-complete in the general case because it can be reduced to the problem of subgraph isomorphism [92], and so the over-all operation is also NP-complete.

Graph Products

We review two common cases of graph products, graph joins and graph composition. In both cases the result contains new edges not found in either input graph.

In graph theory, a graph join of two undirected unlabeled disjoint graphs is defined as the union of the two graphs and additional edges connecting every vertex in G_1 with each vertex in G_2 [47]. An example, adapted for property graphs, is shown in Figure 2.22.

Definition 2.2.11 (Graph join)**.** *A join of graphs G_1 ang G_2 is denoted $G_1 + G_2$ and is defined as:*

$G_1 + G_2 = (V', E', \Pi', \rho', \lambda'_v, \lambda'_e)$,

where

- $N = \{\forall v_1 \in G_1.V \forall v_2 \in G_2.V\, s_e(v_1, \textsf{joins}, v_2), s_e(v_2, \textsf{joins}, v_1)\}$,
- $V' = G_1.V \cup G_2.V$,
- $E' = G_1.E \cup G_2.E \cup N$,
- $\Pi' = G_1.\Pi \cup G_2.\Pi$,
- $\rho' = G_1.\rho + G_2.\rho +$ *total function from* N *to* V',
- $\lambda_v = G_1.\lambda_v + G_2.\lambda_v$,
- $\lambda_e = G_2.\lambda_e + G_2.\lambda_e + \forall e \in N f(e, \textsf{type}) = \textsf{joins}$,
- s_e *is a Skolem function.*

In the case of disjoint graphs, the graph join operation has PTIME data complexity, since it is simply $O(|V_1| \times |V_2|)$. In the case of nondisjoint unlabeled graphs this operation is in NP, because first the graph isomorphism problem must be solved.

SocialScope [6] defines a variant of a graph product they term **composition** that produces a graph induced by new edges composed from edges of the two input graphs. The definition, adjusted for our model of G is as follows:

Definition 2.2.12 (Graph composition)**.** *Graph composition operator* $G_1 \odot_{\sigma 1, \sigma 2, f} G_2$*, where* G_1 *and* G_2 *are graphs,* σ_1 *and* σ_2 *are directional conditions (src, dst), and* g *is a composition function is defined as:*

$G_1 \odot_{\sigma 1, \sigma 2, g} G_2 = (V', E', \Pi', \rho', \lambda'_v, \lambda'_e)$,

where

- $V' = \{u, v\} \mid \forall e \in E', \rho'(e) = (u, v)$,
- $E' = \{s_e(u, \textsf{joins}, v)\} \mid \exists e_1 \in G_1.E \wedge \exists e_2 \in G_2.E s.t. G_1.\rho(e_1).\sigma_1 = G_2.\rho(e_2).\sigma_2 \wedge u = G_1.\rho(e_1).src \wedge v = G_2.\rho(e_2).dst$,
- ρ' *is a total function that maps each edge in* E' *to its source node from* G_1 *and destination node from* G_2,
- $\Pi' = G_1.\Pi \cup G_2.\Pi$,
- $\lambda'_v = G_1.\lambda_v + G_2.\lambda_v$,
- λ'_e *is a partial function that maps an edge in* E' *and property label in* L *to a value determined by input function* g*, based on values of properties of edges* e_1 *and* e_2.

The input composition function g *is similar to a user-defined mapping function* f_e *but accepts two sets of properties as input rather than one.*

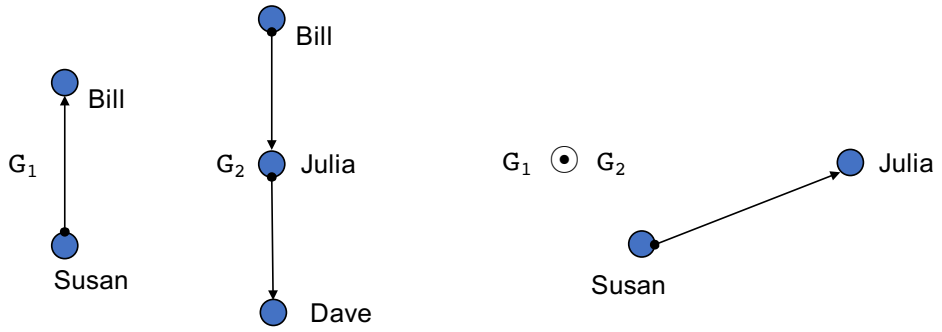


Figure 2.23: Example composition of two graphs G_1 and G_2 with directional condition $\sigma_1 = dst$, $\sigma_2 = src$.

In other words, a new edge is composed from an edge in G_1 and another edge in G_2 if they share a node in common and satisfy the directional conditions. The identities of the new edges are determined by a Skolem function f_e and the properties of new edges are determined by the input function g. Only nodes that appear in new edges are included, i.e., $V' \subseteq G_1.V \cup G_2.V$.

An example is shown in Figure 2.23 where a single edge is generated connecting node Susan to node Julia because of the Susan->Bill edge in G_1 and Bill-Julia edge in G_2.

The complexity of the graph composition problem, when defined thus, is polynomial, since it can be expressed as a join of the two edge sets.

Note that graph composition thus defined is not recursive, but in principle it could be. We could define a variant where new edges are computed based on a regular path query, similar to the node creation case. We return to this point in Section 3.2.9.

Chapter 3: Evolving Graphs Model

Our data model represents a single evolving graph, consisting of nodes, node properties, edges, edge properties, and addition, modification, and deletion of these over time. It uses a closed world semantics – we assume that evolution history is complete, the state of the graph is fully known and recorded at each time instant, and future information is not represented, i.e., there is no uncertain or incomplete information.

In Section 3.1 we define the logical representation of an evolving graph. In Section 3.2 we rigorously define the operations of our Temporal Graph Algebra (TGA). In Section 3.3 we revisit the motivating examples and show how TGA operators can be used to formulate queries to answer them.

3.1 TGraph – Temporal Graph Model

We now describe the logical representation of an evolving graph, called a TGraph. A TGraph represents a single graph, and models the evolution of its topology and of vertex and edge properties. This is in line with the nontemporal graph definition 2.2.1.

In order to represent evolving graphs, we provide two definitions: 1) an extension of Definition 2.2.1, and 2) one that is based on temporal relataional model. The two models are equivalent,

Table 3.1: A co-authorship network represented using the TGraph model, consisting of two nested temporal relations.

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
v2	type=person,name=Bob	[2015/2,2015/5)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

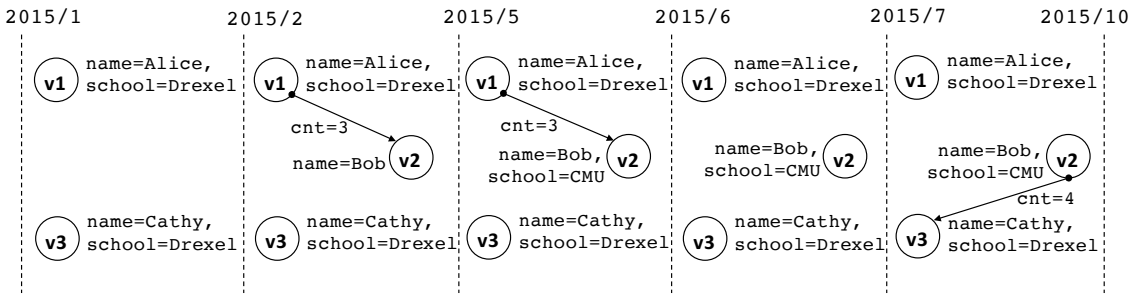


Figure 3.1: Snapshot sequence representation of the example TGraph from Table 3.1. Node and edge type property ommitted for readability.

but it is sometimes easier to define the semantics of an operation over one definition than the other. We extend the property graph model to represent graph nodes and edges as follows: each node and edge is associated with a period of validity, as is each property value. A TGraph can also be represented with two nested temporal relations (Section 2.1.1), associating a fact (existence of a vertex or edge, and an assignment of a value to a vertex or edge attribute) with a time period.

Table 3.1 gives an example of a TGraph that shows evolution of a co-authorship network in the nested temporal relational model. Node 1 persists without change from 2015/1 to 2015/7, but node 2 has a property change event at 2015/5, thus creating a new tuple. For the same of readability we display the nested attribute a as a list of key value pairs.

This TGraph can also be visualized as a sequence of snapshots, as shown in Figure 3.1.

We now give a formal definition of a TGraph.

Definition 3.1.1 (TGraph)**.** *A TGraph $\mathcal{G}$ is a 7-tuple $(V, E, \Pi, \rho, \xi^T, \lambda_v^T, \lambda_e^T)$, where:*

- *V is a finite set of* nodes*,*
- *E is a finite set of* edges*,*
- *Π is a set of available properties, where each element in Π is a pair (L_i, D_i), L_i is a property name, and D_i is the domain of property L_i,*
- *$\rho : E \to (V \times V)$ is a total function that maps an edge to its source and destination node,*
- *$\xi^T : (V \cup E) \times \Omega^T \times \Omega^T \to B$ is a total function that maps a node or an edge and a time period to a boolean, indicating whether the node (resp. edge) existed during that whole time period,*
- *$\lambda_v^T : V \times \bigcup L_i \times \Omega^T \times \Omega^T \to Val$ is a partial function that maps a node from V and a property label from (L_i, D_i) pairs in Π during a time period to a value in domain D_i at that time,*

- $\lambda_e^T : E \times \bigcup L_i \times \Omega^T \times \Omega^T \rightarrow Val$ *is a partial function that maps an edge in* E *and a property label from* (L_i, D_i) *pairs in* Π *during a time period to a value of that property in its domain* D_i *at that time.*

This definition is almost exactly like Definition 2.2.1, with the addition of the ξ^T function that asociates nodes and edges with their periods of validity, and the modification of the λ functions that map from entities to their property values, to include time. Properties can only take on a value during time periods when the node (resp. edge) exists. As in the nontemporal definition, the property set of each node or edge may **not** be empty, since the type property is required. For consistency, we restrict property domains to atomic types and those that can be represented by a V-relation.

As in TRA, time is part of the model rather than an additional property in the property set. Time is treated special and cannot be modified by the user directly, although it can be accessed in predicates. This is one of the lessons from the temporal relational work and we incorporate it here. Note that the ρ function is not temporal – an edge always connects the same two nodes, whenever it exists.

To retrieve individual snapshots we define a get-snapshot operator in the same spirit as the TRA's timeslice operator.

Definition 3.1.2 (Get Snapshot)**.** *Get snapshot operator* snap_p *maps from a* TGraph $\mathcal{G}(V, E, \rho, \xi^T, \lambda_v^T, \lambda_e^T)$ *to a static graph* G *at time point* p*:*

$\mathsf{snap}_p(\mathcal{G}) = (V', E', \Pi, \rho, \lambda_v, \lambda_e)$,

where:

- $V' = \{v \mid v \in V \wedge \exists t(\xi^T(v, t) \wedge p \in t)\}$
- $E' = \{e \mid e \in E \wedge \exists t(\xi^T(e, t) \wedge p \in t)\}$
- $\lambda_v = \lambda_v^T([p, p+1))$
- $\lambda_e = \lambda_e^T([p, p+1))$

As with timeslice, get-snapshot removes all references to time from the TGraph. The set of nodes is limited to those that are valid at point p, as determined by the ξ function. The same with edges. The ξ function itself is not included, since graph snapshots are not temporal and do not contain any time information, per Def. 2.2.1. Both λ functions are obtained through partial application (currying) of the λ^T functions.

In order to take advantage of TRA, we provide an alternate definition using two nested temporal relations, assuming, for the moment, that all node/edge properties are in a single domain D.

Definition 3.1.3 (TGraph-Relational)**.** *A* TGraph *is a pair* $\mathcal{G} = (\mathsf{TV}, \mathsf{TE})$*.* TV *is a valid-time nested temporal relation with schema* $\mathsf{TV}(\underline{v}, a : (K, B) \mid T)$*, where* K *is a property key string, and* B *is a property value .* TV *associates a vertex and its property set with the time period during which it is present and unchanged.* TE *is a valid-time nested temporal relation with schema* $\mathsf{TE}(\underline{e}, v_1, v_2, a : (K, B) \mid T)$*, connecting pairs of vertices from* TV*.*

Relations of $\mathcal{G}$ *must meet the following requirements:*

R1: Unique vertices/ edges *In every snapshot* $\tau_p(\mathsf{TV})$ *and* $\tau_p(\mathsf{TE})$*, where* p *is a time point, a vertex/edge exists at most once. That is, we require set-based semantics with duplicate free temporal relations.*

R2: Referential integrity *In every snapshot* $\tau_p(\mathsf{TE})$ *foreign key constraints hold to* $\tau_p(\mathsf{TV})$ *on both* v_1 *and* v_2*.*

R3: Coalesced *Value-equivalent tuples in all relations of* $\mathcal{G}$ *with consecutive or overlapping time periods are merged. For a nested attribute, value equivalence is a standard set equivalence, i.e., that the sets of property pairs are the same.*

R4: Required property *For any tuple* $v \in \mathsf{TV}, v.a \neq \emptyset$*, and for any tuple* $e \in \mathsf{TE}, e.a \neq \emptyset$*. That is, we require that each node and edge have at least one property (*type*) in its property set.*

R5: Constant edge association *For any pair of tuples* $e_1, e_2 \in \mathsf{TE}, e_1.e = e_2.e \implies e_1.v_1 = e_2.v_1 \wedge e_1.v_2 = e_2.v_2$*. That is, consistent with the non-relational definition, we require that an edge always connects the same two nodes, whenever it exists.*

Property sets are represented by a nontemporal nested relation, and, more specifically, a V-relation (refer to a discussion in Section 2.1.4). Definition 3.1.3 assumes, for simplicity, that all properties draw values from a single domain. It is straight-forward to generalize this definition, allowing the set of property domains to not be limited to a single domain or even to atomic values. We can include any domain that can be represented by an atomic type or a nested relation.

An alternative schema for TV could be $(v, B_1, \ldots, B_n \mid T)$, where n is the size of the set of all properties in $\mathcal{G}$, i.e., $|\Pi|$, and each B_i is an attribute of domain $D_i \in \Pi$, including nested relations. The reason we do not use this model is two-fold:

- The set of all possible properties in $\mathcal{G}$ is large, but most nodes and edges contain only a small subset of the properties. The undefined properties for a specific tuple can be represented by a NULL, but this leads to a very sparse relation.
- Evolving graphs change not only in terms of topology and property values, but by adding new property types. This kind of evolution of the graph requires a modification of the TV

(resp. TE) schema every time a new property is added. In contrast, with the nested model we use, an addition of a new property only requires modification of the schema if the property is of a type not currently present in the set of domains of Π.

The use of nested relations in the schema of TV and TE does not change the expressive power beyond that of flat relations, as shown in [3].

Requirements **R1 and R2** guarantee soundness of the TGraph data structure, ensuring that every snapshot of a TGraph is a valid graph. If we remove requirement **R1**, a snapshot at some point p may contain two instances of the same node or edge, which breaks the set-based graph semantics. Requirement **R2** prevents a situation where an edge connects a node that does not exist at that time instant. Requirement **R3** avoids semantic ambiguity and ensures correctness of algebraic operations in point-stamped temporal models such as ours [53], as we showed in Section 2.1.1. Requirement **R4** provides compatability with the graph-based definition, where property type is required, and avoids any loss of information when relations TV and TE are unnested during operations.

In Figure 3.1 we visualized the example TGraph as a sequence of snapshots. However, we do not model an evolving graph as a sequence of snapshots, as many others have done [18, 22, 36, 38, 55, 57, 63, 65, 77, 81, 84, 94]. Recollect that TRA operators are not defined over a sequence of relation snapshots, because this violates the extended snapshot reducibility property (Section 2.1.1). The same applies here. We discuss this in more detail in Section 6.1.

In the TGraph representation of Definition 3.1.3, vertices and edges are represented as two relations, and vertex and edge attributes are stored as collections of properties. That said, Definition 3.1.3 presents a *logical* data structure that admits different physical representations, including, e.g., a nested node/edge structure, a columnar representation of attributes (each property in a separate relation, supporting different change rates), by a hash-based representation of [86], or in some other way. We explore this further in Chapter 4.

The get-snapshot operator over the TGraph-Relational model is defined by using the timeslice relational operator:

Definition 3.1.4 (Get Snapshot – Relational)**.** *Get snapshot operator over the relational representation of $\mathcal{G}$ is defined as:*

$$\mathit{snap}_p(\mathcal{G}) = (\tau_p(\mathit{TV}), \tau_p(\mathit{TE}))$$

To obtain a graph snapshot over TGraph^R we simply use the TRA timeslice operator τ for both TV and TE relations.

Definitions 3.1.1 and 3.1.3 are equivalent. We prove this by construction.

Theorem 1. *Let $\mathcal{TG}$ be the set of valid TGraphs (per Def. 3.1.1) and let $\mathcal{TG}^{\mathcal{R}}$ be the set of valid TGraph-Relations (per Def 3.1.3). There is a bijection between $\mathcal{TG}$ and $\mathcal{TG}^{\mathcal{R}}$.*

Proof. We prove this theorem in two parts. In the first part, we show that, given $\mathcal{G} \in \mathcal{TG}$, we can construct a $\mathcal{G}^{\mathcal{R}} \in \mathcal{TG}^{\mathcal{R}}$ in polynomial time. In the second part, we show that, give $\mathcal{G}^{\mathcal{R}} \in \mathcal{TG}^{\mathcal{R}}$, we can construct a $\mathcal{G} \in \mathcal{TG}$ in polynomial time.

Part 1:

1. Assume that T is a set of all time instances in $\mathcal{G}$. For each node $v \in V$ and property in Π, we retrieve the set of property values and their periods of validity by applying function λ_v^T to each point in Ω^T. The input size is $|V|$. The output size is at most $|V| \times |T| \times |\Pi|$ if every node exists at every time instant of T and has every property in Π. Recollect that the time domain is of limited precision and we use point semantics, so we can probe every point once. The result is a relation instance $\mathbf{b}$ with schema $(v, K, B \mid T)$, where K is the property name and B is the property value.
2. Nest K and B into a single nested attribute a. The result is a relation with the same schema as TV and of size at most $|V| \times |T|$. Coalesce the result, which can be done in polynomial time [13].
3. For each edge $e \in E$ retrieve the set of intervals of validity and property sets in the same fashion as in points 1 and 2 above, but using function λ_e^T. The result is a relation instance $\mathbf{d}$ with a schema $(e, a|T)$, where a is the property relation. As in points 1-2, the input size is $|E|$ and the output size is $|E| \times |T|$.
4. For each edge $e \in E$ retrieve the source and destination nodes using function ρ. The result is a nontemporal relation instance $\mathbf{f}$ with a schema e, v_1, v_2. The input size and the output size are the same: $|E|$.
5. $\mathsf{TE} = \mathbf{d} \bowtie_e^T \mathbf{f}$. We join relation instances $\mathbf{d}$ and $\mathbf{f}$ by key, and the size of the result is $|\mathbf{d}|$, as there is a single tuple per edge in $\mathbf{f}$ (edge source and destination points cannot change through time).

The above steps may be simplified if the set of validity periods of the functions ξ^T, λ_v^T, and λ_e^T are directly available.

Part 2:

1. The set of nodes V is a result of nontemporal projection of TV onto key v: $V = \pi_v(\mathsf{TV})$. The input size has an upper bound of $|V| \times c$, where c is the maximum number of changes of

any $v \in V$. The output size is $|V|$. Note: this can be expressed solely in TRA as a union of timeslices over temporal projection onto v: $V = \forall p \in T \bigcup \tau_p(\pi_v^T(\mathsf{TV}))$.

2. The set of edges E is a result of nontemporal projection of TE onto key e: $E = \pi_e(\mathsf{TE})$. The output size is $|E|$.
3. The set of available properties $\Pi = \pi_k(\mu^T(\mathsf{TV})) \cup \pi_k(\mu^T(\mathsf{TE}))$. The input size is the total number of node tuples, upper-bounded by $|V| \times c$. Unnesting TV has an upper bound of size $|V| \times c \times |\Pi|$ if every node has every property in Π. The same applies for TE. The final nontemporal projection leads to all the properties available, at most $|\Pi|$.
4. Function $\rho(e_1)$ is computed by a key lookup on e_1 in TE, i.e., $\pi^T_{v_1,v_2}(\sigma^T_{e=e_1}(\mathsf{TE}))$. The size of the output is either 0 or 1, since edge end points cannot change with time.
5. Function ξ^T is computed by a selection query, followed by a slice on TV for nodes (TE for edges resp.): $\xi^T(v_1, t) = \tau_t(\sigma^T_{v=v_1,T=t}(\mathsf{TV})), \xi(e_1, t) = \tau_t(\sigma^T_{e=e_1,T=t}(\mathsf{TE}))$. The size of the output is either 0 or 1.
6. Function λ_v^T is computed by a selection query with an equality predicate on k on TV, followed by a slice: $\lambda_v^T(v_1, l, t) = \tau_t(\sigma^T_{K=l}(\mu^T(\sigma^T_{v=v_1,T=t}(\mathsf{TV}))))$. The size of the first selection is at most 1, since only one tuple may exist in TV for a node at any given time (per Def. 3.1.3, requirement R1). The output of unnesting is of size at most $|\Pi|$ if node v_1 has all the possible properties in Π. Finally, the output of the last selection is of size at most 1 if the node v_1 has property l.
7. Function λ_e^T is computed by a selection query with a temporal predicate and an equality predicate on k on TE, followed by a slice: $\lambda_e^T(e_1, l, t) = \tau_t(\sigma^T_{K=l}(\mu^T(\sigma^T_{e=e_1,T=t}(\mathsf{TE}))))$. The same argument applies as in point 6.

□

When we define operations of our graph algebra, we will use either TGraph or TGraph-relational model, as convenient.

3.2 Temporal Graph Algebra

This section defines our proposed Temporal Graph Algebra, or TGA for short. Our goal is to enable a wide range of analyses useful to potential users, with a focus on analysis over time, and motivated by our examples and those we found in the literature.

The semantics of each operator is specified either by a translation into a sequence of TRA operators, as defined in Section 2.1.3, or by applying navigational patterns from Definition 2.2.7,

extended with time. We assume that the model properties (snapshot reducibility, extended snapshot reducibility, coalescing) are maintained by the temporal relational model, e.g., that if a TRA operator is known to produce uncoalesced results, they are coalesced in the final output, and that the integrity constraints from TE to TV are enforced. We use the foreign key constraint enforcement method that allows the operation and then modifies the output to remove (or trim) tuples from TE for which a corresponding entry does not exist in TV. For readability sake, we use a shorthand notation and refer to $\mathcal{G}$.TV as TV and to $\mathcal{G}$.TE as TE.

But first we need to introduce a new primitive **resolve**, itself a sequence of TRA operators. Recollect that in TRA tuples have no identity and are coalesced only if they are value-equivalent (see Definition 2.1.1 and the subsequent discussion on coalescing). TGA, however, while using temporal valid-time relations, includes both node and edge identity. Identifiers are required in order to distinguish between nodes, resp. edges, as their properties may change over time. In this sense nodes and edges are akin to objects in an object database. We refer to node tuples, resp. edge tuples, that have the same identifiers, as identity-equivalent:

Definition 3.2.1 (Identity-equivalent)**.** *Two node tuples in TV, resp. edge tuples in TE, are* **identity-equivalent** *if they agree on the key – $\underline{v}$ for TV, $\underline{e}$ for TE, regardless of the values of the rest of the attributes.*

To produce a valid TGraph, we need to output valid *keyed* relations TV and TE with no identity-equivalent nodes or edges over overlapping time instants. TRA does not have a mechanism for coalescing based on key only. In addition, TV and TE are *nested* relations, since each node and edge attribute is a set of key-value pairs. Thus, any TGA operation that may produce multiple identity-equivalent nodes or edges over overlaping time periods in its intermediate result requires an additional user input, which is a set of aggregate functions, to be used in the resolve primitive.

Definition 3.2.2 (Resolve primitive)**.** *Resolve primitive $\mathcal{R}(f_1(k_1), \ldots, f_n(k_n), \mathbf{r})$ where id is the nontemporal key of the input relation $\mathbf{r}$ (e.g., v for TV or e for TE), $k_1, \ldots, k_n$ are property names, and $f_1, \ldots, f_n$ are aggregate functions over those properties is defined as:*

$\mathcal{R}(f_1(k_1), \ldots, f_n(k_n), \mathbf{r}) = \nu^T_{a,k,b}(\bigcup_{id} \gamma^T_{f_i(k_i)}(\pi^T_{k=k_i}(\mu^T_a(\mathbf{r}))))$

In other words, resolve unnests (Definition 2.1.26) the input relation, computes a temporal aggregation (Definition 2.1.19) by key, applying specified aggregate function to each property, and then nests (Definition 2.1.25) to produce the final result consistent with the TGraph model.

Table 3.2: Example nested temporal relation **w**.

	v	a		T
		S	C	
w_1	10	100	Susan	[2000/1,2010/1)
w_2	10	20	Sue	[2008/1,2011/1)
w_3	20	5	John	[2010/1,2015/2)

Table 3.3: $\mathcal{R}(max(S), count(C), \mathbf{w})$

v	a		T
	S	C	
10	100	1	[2000/1,2008/1)
10	100	2	[2008/1,2010/1)
10	20	1	[2010/1,2011/1)
20	5	1	[2010/1,2015/2)

Table 3.4: Example TGraph $\mathcal{G}$.

TV

	v	**a**	**T**
$x1$	v1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
$x2$	v2	type=person,name=Bob	[2015/2,2015/5)
$x3$	v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
$x4$	v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

	e	**v1**	**v2**	**a**	**T**
$y1$	e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
$y2$	e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

Example 24. *Consider a nested temporal relation* **w** *in Table 3.2. Tuples* w_1 *and* w_2 *are identity-equivalent and have overlapping time periods. However, they are not value-equivalent, so this relation is considered coalesced in TRA.* $\mathcal{R}(max(S), count(C), \mathbf{w})$ *computes a new relation* $\mathbf{w}'$ *shown in Table 3.3.* $\mathbf{w}'$ *does not have any identity-equivalent tuples, and resolving a conflict in tuple* w_1 *and* w_2 *generates three tuples with non-overlapping timestamps.*

With this foundation we can now define our TGA operators. TGA is snapshot reducible and extended snapshot reducible with respect to the nontemporal graph query language in Section 2.2.2. We return to this more formally in Section 3.4.2.

3.2.1 Trim

As we showed in the motivating examples, it is useful to focus analysis on a particular portion of the overall TGraph history. The trim operator computes a new TGraph, limited only to those nodes and edges that existed during the specified period.

Example 25. *Consider the example TGraph* $\mathcal{G}$ *in Table 3.4, which we use throughout this section. trim with an input period [2015/5,2015/8) computes* $\mathcal{G}'$ *as depicted in Table 3.5. Observe that tuple* $x2$ *is not present because it is wholly outside of the input period. Observe also that the period of validity of tuple* $x1$ *has been modified to equal the intersection with the input period, i.e., it has been trimmed.*

Table 3.5: $\mathsf{trim}^{T}_{[2015/5,2015/8)}(\mathcal{G})$

TV		
v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/5,2015/7)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/8)
v3	type=person,name=Cathy,school=Drexel	[2015/5,2015/8)

TE				
e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/5,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/8)

Definition 3.2.3 (Trim)**.** *The trim operator, denoted* $\mathsf{trim}^{T}_{\boldsymbol{c}}(\mathcal{G})$*, where* $\boldsymbol{c}$ *is a time interval and* $\mathcal{G}$ *is a* TGraph *is defined over the TGraph-Relational representation as:*

$\mathsf{trim}^{T}_{\boldsymbol{c}}(\mathcal{G}) = (TV', TE') \mid TV' = \pi_{v,T \text{ intersect } \boldsymbol{c},a}(\sigma^{T}_{p \text{ overlaps } \boldsymbol{c}}(TV)) \wedge TE' = \pi_{e,v_1,v_2,T \text{ intersect } \boldsymbol{c},a}(\sigma^{T}_{p \text{ overlaps } \boldsymbol{c}}(TE)$

trim is essentially a temporal selection over TV and TE relations to contain only those nodes and edges whose periods have a non-empty intersection with $\boldsymbol{c}$, with their periods trimmed to be within $\boldsymbol{c}$. The selection can be performed over TV and TE relations in any order.

3.2.2 Map

To allow manipulation of node and edge attributes we introduce **vertex-map** and **edge-map** operators. Vertex-map and edge-map apply user-defined map functions to attributes in the same spirit as map in functional languages and the relational projection operator in TRA (Def. 2.1.18).

While the map functions are arbitrary user-specified functions, there are some common cases. Map may specify the set of properties to project out or retain, it may aggregate (e.g., COUNT) or values of a collection property, or unnest a nested value in a property. In other words, mapping is on an entity-by-entity, tuple-by-tuple basis. The time period can be referenced in the mapping function but cannot be changed, that is, the period of validity remains unchanged, consistent with temporal projection in Definition 2.1.18.

Example 26. *Consider again* TGraph $\mathcal{G}$ *in our running example from Table 3.4.* vertex-map *with a projection of the* type *and* name *properties results in a new* TGraph*, depicted in Table 3.6. Observe that tuples* $x2$ *and* $x3$ *generate only a single tuple over the combined time period due to coalescing.*

Definition 3.2.4 (vertex-map)**.** *The vertex-map operator, denoted* $\mathsf{map}^{T}_{v}(f_v, \mathcal{G})$*, where* f_v *is a user-defined mapping function, is defined as:*

Table 3.6: $\mathsf{map}_v^T(\pi_{type,name}, \mathcal{G})$

TV

v	**a**	**T**
v1	type=person,name=Alice	[2015/1,2015/7)
v2	type=person,name=Bob	[2015/2,2015/10)
v3	type=person,name=Cathy	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

Table 3.7: map_e^T(cnt as count when p overlaps [2015/3,2015/6), $\mathcal{G}$)

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
v2	type=person,name=Bob	[2015/2,2015/5)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,count=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

$$\mathsf{map}_v^T(f_v, \mathcal{G}) = (TV', TE') \mid TV' = \{(v, a'|T) \mid (v, a|T) \in \mathsf{TV} \wedge a' = f_v(v, a|T)\} \wedge TE' = TE$$

Definition 3.2.4 is formulated over the TGraph-Relational representation for simplicity, but can be rephrased over TGraph representation (Def. 3.1.1), similarly to the nontemporal definition 2.2.9. vertex-map is essentially a temporal projection over TV that retains all attributes of TV but changes the nested attribute a. Because node periods are not modified, no changes to TE are made to enforce referential integrity. The TV′ relation must be coalesced, which is done automatically by the model.

Similarly:

Definition 3.2.5 (edge-map)**.** *The edge-map operator, denoted* $\mathsf{map}_e^T(f_e, \mathcal{G})$*, where* f_e *is a user-defined mapping function, is defined as:*

$\mathsf{map}_e^T(f_e, \mathcal{G}) = (TV', TE') \mid TV' = TV \wedge TE' = \{(e, v1, v2, a'|T) \mid (e, v1, v2, a|T) \in \mathsf{TE} \wedge a' = f_e(e, v1, v2, a|T)$

Example 27. *The mapping function can refer to the timestamp, as can be seen in Table 3.7, where the mapping function renames the* cnt *property to* count *but only for edges in the spring. Thus, tuple* $y1$ *is changed, while tuple* $y2$ *remains unchanged.*

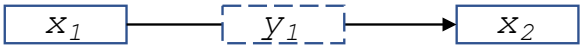


Figure 3.2: Navigational graph pattern P_1, restricting to nonisolated nodes only.

Table 3.8: subgraph^T with pattern P_1 returns a graph with no isolated nodes.

TV

v	a	T
v1	type=person,name=Alice,school=Drexel	[2015/2,2015/6)
v2	type=person,name=Bob	[2015/2,2015/5)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/7,2015/10)

TE

e	v1	v2	a	T
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

3.2.3 Subgraph

Temporal subgraph matching is a generalization of subgraph matching for non-temporal graphs as defined in Definition 2.2.4. Subgraph returns a TGraph matching the input navigational graph pattern (Definition 2.2.7) that may include temporal predicates.

Example 28. *Consider TGraph $\mathcal{G}$ from our running example (Table 3.4). Let's say that we are only interested in non-isolated nodes, i.e., only those that have a non-zero degree. Pattern P_1, shown in Figure 3.2, yields new TGraph $\mathcal{G}$' depicted in Table 3.8. Observe that while the number of node tuples in the output is the same as in the input, their periods are smaller to only include times when edges are present. For instance, $x1$ period has been reduced from [2015/1,2015/7) to [2015/2,2015/6), since no edge connected to $v1$ exists in either [2015/1,2015/2) or [2015/6,2015/7).*

To state this formally, we first modify the definition of a NGP(Definition 2.2.7) to include time:

Definition 3.2.6 (Temporal Navigational Graph Pattern)**.** *A temporal navigational graph pattern (TNGP) is a graph $TNGP$, where V is extended with a set of node variables, E is extended with a set of edge variables and regular path expressions, and property names and values are also extended with variables. Any variable expression may have a temporal predicate.*

A match of TNGP in G is restricted to be isomorphic, i.e., one entity, whether node or edge, cannot match different variables. When no temporal predicates are present in the TNGP, it is semantically evaluated as a regular navigational graph pattern over each snapshot of $\mathcal{G}$.

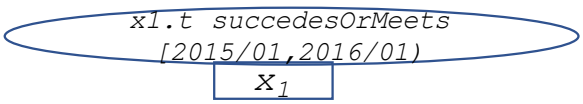


Figure 3.3: Temporal navigational graph pattern restricting nodes by period of validity.

A TNGP may have the following comparators in the expressions: $=, >, <, \leq, \geq, \in, \ni, \subset, \subseteq, \supset, \supseteq$. The set-based comparators are provided for complex-value properties.

Example 29. *Consider a TNGP in Figure 3.3. It matches only nodes with periods of validity since 2016/01. We place the node and edge temporal constraints above the node boxes in the graphical pattern to distinguish them from other types of data.*

We now define $\mathsf{subgraph}^T$ as follows:

Definition 3.2.7 (Temporal subgraph)**.** *The temporal subgraph operator, denoted* $\mathsf{subgraph}^T(P, \mathcal{G})$*, where* q_v *is a set of all node matches of node variables and constants in a temporal navigational graph pattern* P*, and* q_e *is a set of all edge matches of edge variables and constants in* P*, is defined as:*

$$\mathsf{subgraph}^T(P, \mathcal{G}) = (q_v(P, \mathcal{G}), q_e(P, \mathcal{G}), \Pi, \rho, \xi^T, \lambda_v^T, \lambda_e^T)$$

There is a potential semantic ambiguity that does not arise in static graphs. Consider TGraph $\mathcal{G}$ from our running example. Given a simple query on nodes that selects only those that have a school property, connected by edges that persist for longer than 2 month, two results are possible, as depicted in Tables 3.9a and 3.9b. In the first case, the queries on nodes and edges are executed first and then the foreign key constraint is enforced. Thus tuple $y1$ is selected but then its duration is shortened because tuple $x2$ is not selected, and we have an edge in the result that is in fact shorter than 2 months required duration. In the second case, the query on nodes is carried out and the foreign key constraint is enforced before carrying out the query on the edges, and thus tuple $y2$ does not appear in the result.

To resolve this ambiguity we apply the principle of extended snapshot reducibility that states that the presence of temporal predicates should not change the behavior of the temporal operator with respect to the snapshot reducibility in all other ways. Thus neither interpretation of this subgraph query is correct. Another way to think about it is to rephrase the query so: which nodes have the property and are connected continuously for at least two months. Clearly only nodes $v2$ and $v3$ in the period from $[2015/7, 2015/10)$ connected by edge $e2$ meet the pattern.

Table 3.9: Subgraph of nodes with a school property connected by edges that persist for longer than 2 months.

(a) foreign key constraint enforced last

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/5,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

(b) edge predicate applied last

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

Table 3.10: $\mathsf{agg}^T(p_2, G)$

TV

v	a	T
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/2)
v1	type=person,name=Alice,school=Drexel,olde=1	[2015/2,2015/6)
v1	type=person,name=Alice,school=Drexel	[2015/6,2015/7)
v2	type=person,name=Bob,olde=1	[2015/2,2015/5)
v2	type=person,name=Bob,school=CMU,olde=1	[2015/5,2015/6)
v2	type=person,name=Bob,school=CMU	[2015/6,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	v1	v2	a	T
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

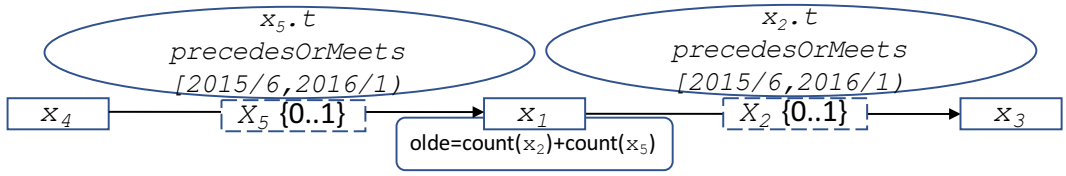


Figure 3.4: TNGP p_2 that computes the degree of each node accounting only for old edges.

3.2.4 Aggregation

Temporal aggregation is a generalization of the graph aggregation operation defined in Section 2.2.2 to include temporal predicates and operate over temporal data. It computes the value of a new node property based on information available at the node itself, at the edges associated with the node, and at its neighbors. Aggregation can be used to compute simple properties such as in-degree of a node, or more complex ones such as the set of countries that the friends of v visited in the past year.

Example 30. *Consider our running example* TGraph $\mathcal{G}$ *(Table 3.4). We compute a new node property* olde *that is the degree of each node counting only edges that existed before 2015/6 using temporal aggregation and pattern p_2, depicted in Figure 3.4. The result is shown in Table 3.10.*

Definition 3.2.8 (Aggregation)**.** *Temporal graph aggregation $\mathsf{agg}^T(P, G)$, where P is a TNGP (Definition 3.2.6) extended with aggregation functions and M is a set of aggregating properties in P is defined as:*

$\mathsf{agg}^T(P, G) = (V, E, \Pi', \rho, \xi^T, \lambda_v^{T'}, \lambda_e^{T'})$*, where*

- $\Pi' = \Pi \cup M$,
- $Dom(\lambda'_v) = Dom(\lambda_v) \cup A_v$,

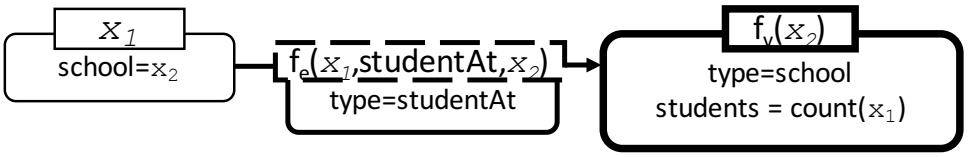


Figure 3.5: TNGP to create nodes for each value of school property.

- $Dom(\lambda'_e) = Dom(\lambda_e) \cup A_e$,
- A_v *is the set of values of all node aggregating expressions in* P *applied to* G,
- A_e *is the set of values of all edge aggregating expressions in* P *applied to* G.

Aggregation can be used to compute the lengths of *journeys*. A journey is transitive closure on the graph conditional on the time property of the edges – two nodes are connected by a new edge if there is a non-decreasing time path between them [20, 38].

Example 31. *Consider TGraph* $\mathcal{G}$ *in our running example. We can compute the longest journey from each node in* $\mathcal{G}$ *using aggregation. The longest journey from node* $v1$ *is of length 2 through node* $v2$ *to node* $v3$*, since edge tuple* $y2$ *starts later than tuple* $y1$*. However, node* $v1$ *is unreachable from node* $v3$ *even if we reverse the edge direction because that requires traveling back in time.*

3.2.5 Node Creation

The node creation operator enables the user to analyze an evolving graph at different levels of granularity. This operator comes in two variants — based on node attributes or based on temporal window.

Attribute-based Node Creation

Attribute-based node creation is a temporal generalization of the graph node creation operation in Definition 2.2.10. Recall that node creation adds new nodes that represent a matching input pattern. Attribute-based node creation takes in a TNGP extended with Skolem functions, but is otherwise the same as the nontemporal one.

Example 32. *Consider our running example TGraph* $\mathcal{G}$ *(Table 3.4). We can create new summary nodes to represent each school based on the* school *property of the nodes using the pattern in Figure 3.5. The result is shown in Table 3.11, with two new nodes* Drexel *and* CMU *and* studentAt *edges connecting to these nodes. The TNGP can specify aggregation functions to compute properties of the new nodes based on the input pattern and here* count *is used to generate the* students *property.*

Table 3.11: Attribute-based node creation based on property school.

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
v2	type=person,name=Bob	[2015/2,2015/5)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)
Drexel	type=school,students=2	[2015/1,2015/7)
Drexel	type=school,students=1	[2015/7,2015/10)
CMU	type=school,students=1	[2015/5,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)
e3	v1	Drexel	type=studentAt	[2015/1,2015/7)
e4	v3	Drexel	type=studentAt	[2015/1,2015/10)
e5	v2	CMU	type=studentAt	[2015/5,2015,10)

Definition 3.2.9 (Attribute-based node creation)**.** *Attribute-based node creation, denoted* $\mathsf{node}_a^T(P^s, \mathcal{G})$ *where* P^s *is a TNGP extended with Skolem functions, is defined as:*

$\mathsf{node}_a^T(P^s, \mathcal{G}) = (V \cup M, E \cup N, \Pi, \rho', \xi^{T'}, \lambda_v^{T'}, \lambda_e^{T'})$*, where*

- M *is a set of new nodes created by the Skolem function* s_v *based on the matches of the variables used as parameters to* s_v *in* P^s*,*
- N *is a set of new edges created by the Skolem function* s_e *based on the matches of the variables used as parameters to* s_e *in* P^s*,*
- ρ' *is extended to include mappings for new edges in* N*,*
- $\xi^{T'}$ *is extended to include mappings for new nodes and edges in* M *and* N*,*
- $\lambda_v^{T'}$ *is extended to include properties for new nodes in* M*,*
- $\lambda_e^{T'}$ *is extended to include properties for new edges in* N*.*

Intuitively, like in the non-temporal version, the nodes are computed by applying the pattern. Every new node is assigned an identity by a Skolem function. Each new node is connected to the pattern from which it originated with new edges, the identity of which is also assigned by a Skolem function. The new nodes and edges are added to the input TGraph.

Attribute-based node creation allows the user to, for instance, generate a TGraph in which nodes correspond to disjoint groups of nodes in the input that agree on the values of all grouping attributes. It can also be used with more complex patterns, such as to generate a new node for each connected component and assign it a size property.

Table 3.12: Node creation with a 3-month window.

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
v2	type=person,name=Bob	[2015/1,2015/4)
v2	type=person,name=Bob,school=CMU	[2015/4,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/1,2015/7)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

Window-based Node Creation

It is interesting and insightful to analyze an evolving graph at different levels of temporal granularity. For example, the user may want to redefine temporal resolution and look at the graph at that scale, irrespective of whether this resolution happens to be finer or coarse than the natural evolution rate of the graph. For this, we introduce a window-based node creation operator that is similar to the *moving window temporal aggregation* in temporal relational algebra. Our approach is inspired by stream aggregation work of [68], adopted to graphs, and by generalized quantifiers of [50].

Window-based node creation modifies tuple periods based on consecutive temporal windows from the window specification, such as 2 months or 10 years.

Example 33. *Consider again TGraph $\mathcal{G}$ in our running example depicted in Table 3.4. Table 3.12 shows the result of applying node creation with a 3-month window. Note that the period of node v2 in the result starts earlier, on 2015/1, because the node existed in the first defined window [2015/1,2015/4). It also ends earlier because two versions existed during the second defined window [2015/4,2015/7) and we picked the last.*

The above example essentially declared a node or an edge to be valid for each time window if it existed during any part of that window. It is useful to be able to quantify required node/edge duration in order to consider it valid. We use *quantifiers* for this purpose. Node and edge quantifiers r_v and r_e are of the form { all | most | at least n | exists }, where n is a decimal representing the percentage of the time during which a node or an edge existed, relative to the duration of the window. Quantifiers are useful for observing different kinds of temporal evolution, e.g., to observe

only strong connections over a volatile evolving graph, we may want to only include nodes that span the entire window ($r_v = \mathsf{all}$), and edges that span a large portion of the window ($r_e = \mathsf{most}$).

Window specification is of the form n $\{unit|\mathsf{changes}\}$, where n is an integer, and $unit$ is a time unit, e.g., 10 min, 3 years, or any multiple of the usual time units. When the window specification is in the form n changes, it defines the window by the number of changes that occurred in $\mathcal{G}$ (affecting any of its constituent relations). Window boundaries are defined left-to-right, i.e., from least to most recent.

Window specification generates a temporal relation W with the schema $(N|T)$, where each tuple associates a window number with its duration.

Putting it all together:

Definition 3.2.10 (Window-based node creation)**.** *The window-based node creation operator, denoted* $\mathsf{node}^T_w(w, r_v, r_e, f_{v_1}(k_1), \ldots, f_{v_n}(k_n), f_{e_1}(l_1), \ldots, f_{e_m}(l_m), \mathcal{G})$*, where w is the window specification, W is the relation generated by w consisting of periods defined by the specification, r_v and r_e are node and edge quantifiers, and each $f_{v_j}(k_j)$ ($f_{e_j}(l_j)$) specifies an aggregation function f_{v_j} (resp. f_{e_j}) to be applied to a node property k_j (resp. edge property l_j) is defined as:*

$$\mathsf{node}^T_w(w, r_v, r_e, f_{v_1}(k_1), \ldots, f_{v_n}(k_n), f_{e_1}(l_1), \ldots, f_{e_m}(l_m), \mathcal{G}) =$$
$$\{TV', TE' \mid TV' = \sigma^T_{r_v}(\mathcal{R}(f_{v_1}(k_1), \ldots, f_{v_n}(k_n), \pi^T_{v,a}(TV \times^T W)))\wedge$$
$$TE' = \sigma^T_{r_e}(\mathcal{R}(f_{e_1}(l_1), \ldots, f_{e_m}(l_m), \pi^T_{e,v1,v2,a}(TE \times^T W)))\}$$

Essentially, we compute a cross product of each node and edge to every window it overlaps, project out the dummy window number, apply the resolve primitive for cases where more than one identity-equivalent tuple is in the same window, and select only those that meet the quantification. It does not matter in which order computation is applied, i.e., either TV or TE can be computed first, as long as the foreign key constraint is enforced in the final result.

For the purposes of this operator we expand the list of supported aggregation functions to include temporal first ($min(t_1.s, t_2.s)$) and last ($max(t_1.s, t_2.s)$).

Our window specification by change is similar to slide-by-row window in stream aggregation [68]. Note that, because TGraph algebra is compositional, we do not support node creation with overlapping windows, because it does not produce a valid TGraph. To see why this is so, consider applying a sliding window of 3 months range with 1 month slide to graph T in our running example TGraph $\mathcal{G}$. We would produce the following tuples for v_1: $(v_1, [2015/1, 2015/4), a_1)$, $(v_1, [2015/2, 2015/5), a_2)$,

Table 3.13: $\mathsf{node}_w^T(r_v = \mathsf{always}, r_e = \mathsf{exists}, f_{v_1} = \mathsf{first}(\mathsf{name}), f_{v_2} = \mathsf{first}(\mathsf{school}), \mathcal{G})$

W

N	**T**
1	[2015/1,2015/6)
2	[2015/6,2015/10)

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/6)
v2	type=person,name=Bob,school=CMU	[2015/6,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/5,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/6,2015/10)

$(v_1, [2015/3, 2015/6))$, and so on, which clearly violates the temporally coalesced requirement in definition 3.1.3.

Example 34. *Table 3.13 illustrates window-based node creation by change ($w = 3$ changes) with all quantifier for nodes and exists for edges, and first aggregation function for node and edge properties. v_2 is present in the result starting at 5/15 because it did not exist for the entirety of the first window. Edge id $e1$ is reduced in duration because before 2015/5 $v2$ does not exist and after 2015/6 $v1$ does not exist. On the other hand edge id $e2$ is extended in duration to cover the whole second window as both of its end points exist and it existed at some point during the window in the input.*

3.2.6 Union

We support temporal versions of the three binary set operators intersection ($\cap^{TG}$), union ($\cup^{TG}$), and difference ($\setminus^{TG}$). The union operator produces a TGraph that contains nodes and edges that exist in either $\mathcal{G}_1$ or $\mathcal{G}_2$.

Example 35. *Consider another TGraph $\mathcal{G}_2$ depicted in Table 3.14. Temporal union of $\mathcal{G}$ and $\mathcal{G}_2$ is shown in Table 3.15. As with the TRA $\cup^T$ (Def.2.1.22), each node (resp. edge) holds at every time instant in which it holds in either of the input TV (resp. TE) relations. Thus node $v1$ period is extended based on the union of tuples $x1$ and $w1$.*

There is a graph-specific difference between TRA union and TGA union related to identity, namely, there is no guarantee that a node or an edge with the same identity and at the same time instant has the same property set in both input graphs. Previously published work either avoided

Table 3.14: Example TGraph $\mathcal{G}_2$.

	TV				
	v			**a**	**T**
$w1$	v1			type=person,name=Alice,school=Drexel	[2015/2,2015/9)
$w2$	v2			type=person,name=Bob	[2015/2,2015/4)
$w3$	v4			type=person,name=Dave,group=AI	[2015/2,2015/6)
	TE				
	e	**v1**	**v2**	**a**	**T**
$\hat{e}1$	e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/4)
$\hat{e}2$	e3	v4	v2	type=co-author,cnt=2	[2015/4,2015/6)

Table 3.15: Union of $\mathcal{G}$ and $\mathcal{G}_2$.

TV			
v		**a**	**T**
v1		type=person,name=Alice,school=Drexel	[2015/1,2015/9)
v2		type=person,name=Bob	[2015/2,2015/5)
v2		type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3		type=person,name=Cathy,school=Drexel	[2015/1,2015/10)
v4		type=person,name=Dave,group=AI	[2015/2,2015/6)
TE			
e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)
e3	v4	v2	type=co-author,cnt=2	[2015/4,2015/6)

Table 3.16: Example TGraph $\mathcal{G}_3$.

	TV			
	v		**a**	**T**
$z1$	v1		type=person,name=Alice,group=DB	[2015/2,2015/9)
$z2$	v2		type=person,name=Bobby	[2015/2,2015/4)
$z3$	v2		type=person,name=Bob,group=DB	[2015/4,2015/6)
$z4$	v4		type=person,name=Dave,group=AI	[2015/2,2015/6)
	TE			
	e	**v1**	**v2**	**a**	**T**
$\bar{e}1$	e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/4)
$\bar{e}2$	e3	v4	v2	type=co-author,cnt=2	[2015/4,2015/6)

this problem by not using the property model (as long as ids match there is no issue) or side-stepped it by requiring that both input graphs are drawn from the same underlying graph [6]. We aim to provide a more general operation without this restriction. To this end we require aggregation functions as additional input to $\cup^{TG}$ to resolve identity-equivalent tuples with overlapping periods.

Definition 3.2.11 (Union)**.**

TGraph union operator, denoted $\cup^{TG}_{f_{v1}(k_1),\ldots,f_{vn}(k_n),f_{e1}(l_1),\ldots,f_{em}(l_m)}$*, where each* $f_{vj}(k_j)$ *(*$f_{ej}(l_j)$*) specifies an aggregation function* f_{vj} *(resp.* f_{ej}*) to be applied to a node property* k_j *(resp. edge property* l_j*) is defined as:*

$\mathcal{G}_1 \cup^{TG}_{f_{v1}(k_1),\ldots,f_{vn}(k_n),f_{e1}(l_1),\ldots,f_{em}(l_m)} \mathcal{G}_2 = (TV', TE') \mid TV' = \mathcal{R}(f_{v1},\ldots,f_{vn}, TV_1 \cup^T TV_2) \wedge TE' = \mathcal{R}(f_{e1},\ldots,f_{em}, TE_1 \cup^T TE_2)$

We use the TRA's $\cup^T$ operator on both constituent relations of $\mathcal{G}$, followed by applying the resolve primitive to coalesce identity-equivalent tuples. With this approach nodes (resp. edges) can have different property sets while still resulting in a valid output TGraph. Note, however, that it is still necessary for the node (resp. edge) identifiers of the two graphs to be from the same universe.

For the purposes of binary operators we expand the set of aggregation functions to include left, which gives precedence to the value of the property from the left operand, and right, which does the opposite. These functions are useful if it is known that one of the two input TGraphs contains the ground truth. The above definition assumes that an aggregation function is provided for each property in Π. In practice, a syntax over TGA can specify a default aggregation function, such as set, so that the user does not have to list all possible properties.

Example 36. *Consider another TGraph* $\mathcal{G}_3$ *in Table 3.16. Union of this graph with* $\mathcal{G}$ *from our running example is shown in Table 3.17. Note that union of tuple* $x2$ *with* $z2$ *produces a new tuple (v2,name=Bob|[2015/2,2015/4)) because precedence is given to the* $\mathcal{G}$*'s* name *property (Bob, not*

Table 3.17: $\mathcal{G} \cup^{TG}_{\mathsf{left(name),max(count)}} \mathcal{G}_3$

TV

v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/2)
v1	type=person,name=Alice,school=Drexel,group=DB	[2015/2,2015/7)
v1	type=person,name=Alice,group=DB	[2015/7,2015/9)
v2	type=person,name=Bob	[2015/2,2015/4)
v2	type=person,name=Bob,group=DB	[2015/4,2015/5)
v2	type=person,name=Bob,school=CMU,group=DB	[2015/5,2015/6)
v2	type=person,name=Bob,school=CMU	[2015/6,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)
v4	type=person,name=Dave,group=AI	[2015/2,2015/6)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)
e3	v4	v2	type=co-author,cnt=2	[2015/4,2015/6)

Bobby) with the left *aggregation function. Also note that four different tuples are produced for* $v2$ *for the four different periods of value equivalence.*

3.2.7 Intersection

The TGraph intersection operator produces a TGraph of nodes and edges that exist in both $\mathcal{G}_1$ and $\mathcal{G}_2$ and is defined in a similar fashion as the union operator:

Definition 3.2.12 (Intersection)**.**

TGraph intersection operator, denoted $\cap^{TG}_{f_{v1}(k_1),\ldots,f_{vn}(k_n),f_{e1}(l_1),\ldots,f_{em}(l_m)}$*, where each* $f_{vj}(k_j)$ *(*$f_{ej}(l_j)$*) specifies an aggregation function* f_{vj} *(resp.* f_{ej}*) to be applied to a node property* k_j *(resp. edge property* l_j*) is defined as:*

$\mathcal{G}_1 \cap^{TG}_{f_{v1}(k_1),\ldots,f_{vn}(k_n),f_{e1}(l_1),\ldots,f_{em}(l_m)} \mathcal{G}_2 = (TV', TE') \mid TV' = \mathcal{R}(f_{v1},\ldots,f_{vn},\pi^T_{v,a_1\cup a_2}(TV_1 \bowtie^T_v TV_2)) \wedge TE' = \mathcal{R}(f_{e1},\ldots,f_{em},\pi^T_{e,v_1,v_2,a_1\cup a_2}(TE_1 \bowtie^T_{e,v_1,v_2} TE_2))$

Instead of using the TRA set intersection operator, we must use an inner join to allow for tuples that are identity-equivalent but have different property sets. If we use set intersection, identity-equivalent tuples that are not value-equivalent would be eliminated. We apply the resolve primitive in the usual manner. There is no dependence between computation of TV and TE– it can be done in any order or in parallel.

Example 37. *Consider again* TGraphs $\mathcal{G}$ *and* $\mathcal{G}_3$*. Their intersection is shown in Table 3.18. Note that neither node* $v3$ *nor* $v4$ *are produced because they only exist in one or the other of the input*

Table 3.18: $\mathcal{G} \cap^{TG}_{\mathsf{left(name),max(count)}} \mathcal{G}_3$

TV

v	a	T
v1	type=person,name=Alice,school=Drexel,group=DB	[2015/2,2015/7)
v2	type=person,name=Bob	[2015/2,2015/4)
v2	type=person,name=Bob,group=DB	[2015/4,2015/5)
v2	type=person,name=Bob,school=CMU,group=DB	[2015/5,2015/6)

TE

e	v1	v2	a	T
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/4)

Table 3.19: $\mathcal{G} \setminus^{TG} \mathcal{G}_3$

TV

v	a	T
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/2)
v2	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	v1	v2	a	T
e1	v1	v2	type=co-author,cnt=3	[2015/4,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)

TGraphs but not both. Similarly for edges $e3$ and $e4$. Period (2015/2,2015/4) for $v2$ is computed as a result of the join of [2015/2,2015/5) in $\mathcal{G}$ and [2015/2,2015/4) in $\mathcal{G}_3$.

3.2.8 Difference

The difference operator produces nodes and edges that exist in $\mathcal{G}_1$ but not in $\mathcal{G}_2$. It does not require aggregation functions for resolution but is otherwise defined in a similar manner:

Definition 3.2.13 (Difference)**.**

TGraph difference operator, denoted $\setminus^{TG}$ is defined as:

$\mathcal{G}_1 \setminus^{TG} \mathcal{G}_2 = (TV', TE') \mid TV' = \pi^T_{TV_1.v,TV_1.a}(\sigma^T_{TV_2.a isNULL}(TV_1 \ltimes^T_v TV_2)) \wedge TE' = \pi^T_{TE_1.e,TE_1.v1,TE_1.v2,TE_1.a}(\sigma^T_{TE_2.a isNULL}(TE_1 \ltimes^T_{e,v_1,v_2} TE_2))$

Since we operate based on node (resp. edge) identity rather than the tuple equivalence, instead of the TRA's difference operator we use a left outer join, select only those tuples that exist in $\mathcal{G}_1$ but not in $\mathcal{G}_2$ using the $notNULL$ predicate, and then project out the extra columns.

Example 38. *To continue the previous example, consider $\mathcal{G}$ and $\mathcal{G}_3$. The result of $\mathcal{G} \setminus^{TG} \mathcal{G}_3$ is shown in Table 3.19. Tuple for $v1$ is only produced for the period of [2015/1,2015/2) since at all other points of tuple $x1$ it overlaps with tuple $z2$. Similarly for edge $e1$.*

Table 3.20: Result of edge creation operator over $\mathcal{G}$ and $\mathcal{G}_2$.

TV		
v	**a**	**T**
v1	type=person,name=Alice,school=Drexel	[2015/1,2015/9)
v2	type=person,name=Bob	[2015/2,2015/5)
v2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)
v4	type=person,name=Dave,group=AI	[2015/2,2015/6)

TE				
e	**v1**	**v2**	**a**	**T**
e1	v1	v2	type=co-author,cnt=3	[2015/2,2015/6)
e2	v2	v3	type=co-author,cnt=4	[2015/7,2015/10)
e3	v4	v2	type=co-author,cnt=2	[2015/4,2015/6)
e4	v1	v4	type=joined	[2015/2,2015/6)
e5	v2	v1	type=joined	[2015/2,2015/9)
e6	v2	v4	type=joined	[2015/2,2015/6)
e7	v3	v1	type=joined	[2015/2,2015/9)
e8	v3	v2	type=joined	[2015/2,2015/4)
e9	v3	v4	type=joined	[2015/2,2015/6)

3.2.9 Edge Creation

Edge creation is a binary operator and is a temporal generalization of the various non-temporal graph products defined in Section 2.2.2. The reason we call it **edge creation** is because the edges produced in the output may not exist in either of the two input graphs.

Example 39. *Consider again example* TGraphs $\mathcal{G}$ *and* $\mathcal{G}_2$*. We can apply the edge creation operator to produce a standard graph join (Def.2.2.11), where each node in* $\mathcal{G}$ *is connected by a new edge to each node in* $\mathcal{G}_2$ *for every time instant the pair exists. The result is shown in Table 3.20. It may be easier to understand the result by visualizing the two input* TGraphs *as sequences of snapshots, as shown in Figure 3.6, where some of the new edges are added in red. For instance, new edge* $e4$ *is produced between nodes* $v1$ *and* $v4$ *for the period [2015/2,2016/6), which is the intersection of the periods of tuples* $x1$ *and* $w3$*.*

The nodes in the two input graphs are combined, while the edges are computed with a temporal navigational graph pattern extended with namespaces.

Definition 3.2.14. *The edge creation operator* $\mathsf{edge}^T_P(\mathcal{G}_1, \mathcal{G}_2)$*, where* N *is a set of edges with an expression containing a Skolem function in a TNGP* P *over* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *extended with namespaces, and each* $f_{vj}(l_j)$ *specifies an aggregation function* f_{vj} *to be applied to a node property* l_j*, is defined as:*

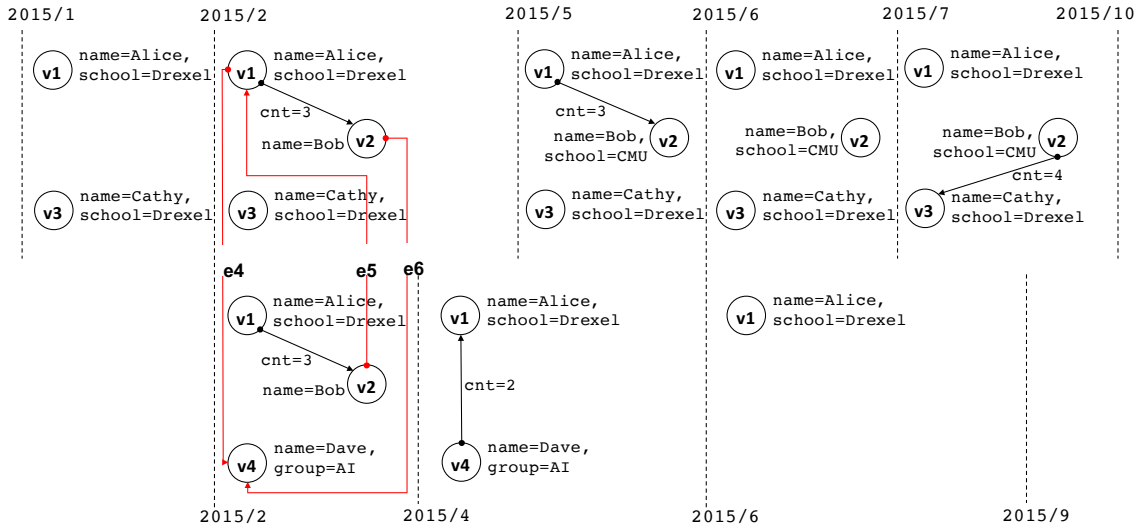


Figure 3.6: Visualization of edge creation over $\mathcal{G}$ and $\mathcal{G}_2$ using snapshots.

$\mathit{edge}^T_{P,f_{v1}(l_1),\ldots,f_{vn}(l_n)}(\mathcal{G}_1,\mathcal{G}_2) = (V', E', \Pi', \rho', \xi^{T'}, \lambda_v^{T'}, \lambda_e^{T'})$, *where:*

- $V' = \mathcal{G}_1.V \cup \mathcal{G}_2.V$,
- $E' = N$,
- $\Pi' = \mathcal{G}_1.\Pi \cup \mathcal{G}_2.\Pi$,
- ρ' *is a total function that maps each new edge in* N *to source and destination nodes in* V',
- $\xi^{T'}$ *is a union of* $\mathcal{G}_1.\xi^T$ *and* $\mathcal{G}_2.\xi^T$, *extended to include mapings to time for new edges in* N,
- $\lambda_v^{T'}(l_i) = f_i(\mathcal{G}_1.\lambda_v^T(l_i), \mathcal{G}_2.\lambda_v^T(l_i))$,
- $\lambda_e^{T'}$ *maps an edge in* N *and a property label from* (L_i, D_i) *pairs in* Π' *during a time period to a value of that property in its domain* D_i *at that time based on the aggregation expressions in* P.

Similar to node creation, a Skolem function is required to assign identity to new edges. The nodes are generated exactly as in the temporal union $\cup^{TG}$. Intuitively, edge creation returns a new TGraph from nodes of $\mathcal{G}_1$ and $\mathcal{G}_2$, connected by edges determined by the input pattern.

Edge creation has several important applications. It can be used to transpose TGraph edges or compute friend-of-friend edges, passing in the same TGraph as both arguments. Since P can be recursive and include predicates over the timestamps, edge^T can create new edges representing journeys. Recall that a journey is a path in the evolving graph with non-decreasing time edges. By adding a temporal condition to P, we can obtain journeys similar to time-concurrent paths.

Example 40. *Consider TGraphs* $\mathcal{G}$ *and* $\mathcal{G}_2$. *To compute all journeys we use edge creation with a temporal navigational path pattern to restrict paths to those that go only forward in time. The*

Table 3.21: $\mathsf{edge}_P^T(\mathcal{G}, \mathcal{G}_2)$

TV			
v		**a**	**T**
v1		type=person,name=Alice,school=Drexel	[2015/1,2015/9)
v2		type=person,name=Bob	[2015/2,2015/5)
v2		type=person,name=Bob,school=CMU	[2015/5,2015/10)
v3		type=person,name=Cathy,school=Drexel	[2015/1,2015/10)
v4		type=person,name=Dave,group=AI	[2015/2,2015/6)
TE			
e	**v1**	**v2**	**a**	**T**
e4	v1	v3	type=co-author,cnt=7	[2015/2,2015/9)
e5	v4	v3	type=co-author,cnt=6	[2015/4,2015/6)

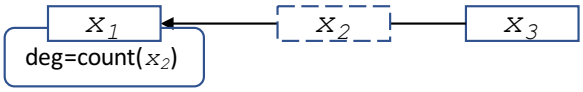


Figure 3.7: Temporal navigational graph pattern p_1 to compute node degrees.

*result is shown in Table 3.21. Note that two new edges are added, e*4 *and e*5*. e*4 *is the result of tuples y*1 *and y*2*, while e*5 *is the result of tuples ê*2 *and y*2*. The property* cnt *on the new edges is computed here with a* SUM *aggregate function in the pattern.*

3.3 Usecases

Now that we have introduced all the operators of the TGA, let us revisit the motivating usecases from Chapter 1. We can express each use case as a sequence of TGA operators.

3.3.1 Vertex Influence over Time

Question: What are the high-influence nodes over the past 5 years, and is their influence persistent over time?

1. Select a subset of the data representing the 5 years of interest, using trim:

$$\mathcal{G}_1 = \mathsf{trim}^T_{[2010,2015)}(wikitalk)$$

2. Compute in-degree (prominence) of each node during each time point using aggregation and pattern p_1 depicted in Figure 3.7:

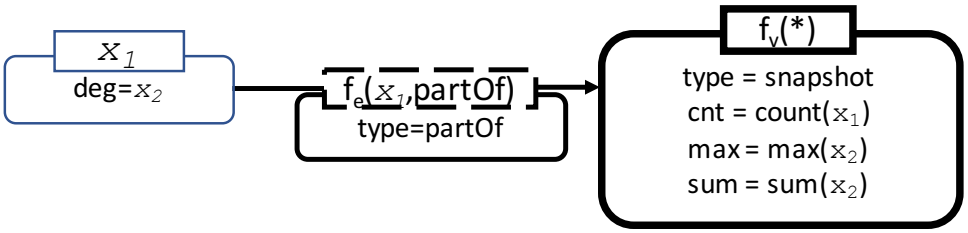


Figure 3.8: Navigational graph pattern p_2.

$\mathcal{G}_2 = \mathsf{agg}^T_{p_1}(\mathcal{G}_1)$

3. Aggregate degree information per node across the timespan of $\mathcal{G}_2$, collecting values into a map using the window-based node creation operator:

$\mathcal{G}_3 = \mathsf{node}^T_w(\mathsf{w} = \mathsf{lifetime}, \mathsf{f_v} = \{\mathsf{map(deg)}\}, \mathcal{G}_2)$

4. Transform the attributes of each node to compute the coefficient of variation from the map of degree values, using the vertex-map operator:

$\mathcal{G}_4 = \mathsf{map}^T_v(\mathsf{f_v} = \mathsf{stdev(deg)/mean(deg)} * 100, \mathcal{G}_3)$

3.3.2 Graph Centrality over Time

Question: How has graph centrality changed over time?

1. Compute a temporally aggregated view of the graph into 2-months windows using the window-based node creation operator with always node and edge quantifiers:

$\mathcal{G}_1 = \mathsf{node}^T_w(\mathsf{w} = 2\ \mathsf{mon}, \mathsf{r_v} = \mathsf{always}, \mathsf{r_e} = \mathsf{always}, wikitalk)$

2. Compute in-degree of each node with the aggregation operator:

$\mathcal{G}_2 = \mathsf{agg}^T_{p_1}(\mathcal{G}_1)$

3. Create a new graph, in which all nodes that are present at a given time point (snapshot) are grouped into a single node with the attribute-based node creation operator, using pattern p_2 in

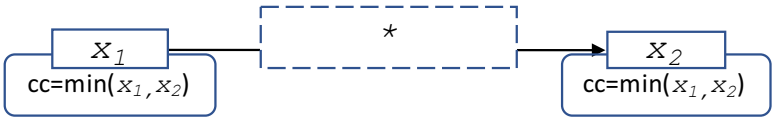


Figure 3.9: Connected components pattern p_3.

Figure 3.8. Accumulate maximum, sum and count of the values of deg as properties at that node.

$\mathcal{G}_3 = \mathsf{node}_a^T(p_2, \mathcal{G}_2)$

4. Compute degree centrality at each time point using the vertex-map operator:

$\mathcal{G}_4 = \mathsf{map}_v^T(\mathsf{f_v} = \mathsf{cent} = (\mathsf{max} * \mathsf{cnt} - \mathsf{sum})/(\mathsf{cnt}^2 - 3 * \mathsf{cnt} + 2), \mathcal{G}_3)$

3.3.3 Communities over Time

Question: In a sparse communication network, on what time scale can we detect communities?

1. Aggregate the graph into 6-month windows:

$\mathcal{G}_1 = \mathsf{node}_w^T(\mathsf{w} = 6\ \mathsf{mon}, \mathsf{r_v} = \mathsf{always}, \mathsf{r_e} = \mathsf{always}, wikitalk)$

2. Compute weakly connected components at each time point using the aggregation operator and a pattern p_3 depicted in Figure 3.9:

$\mathcal{G}_2 = \mathsf{agg}_{p_3}^T(\mathcal{G}_1)$

3. Add a node for each connected component, and compute the size of the connected component, using attribute-based node creation and pattern p_4 (not depicted):

$\mathcal{G}_3 = \mathsf{node}_a^T(p_4, \mathcal{G}_2)$

4. Filter out nodes that represent communities too small to be useful (e.g., of 1-2 people) using vertex-subgraph and a simple pattern p_5 with size>3 predicate (not depicted):

$\mathcal{G}_4 = \mathsf{subgraph}^T(p_5, \mathcal{G}_3)$

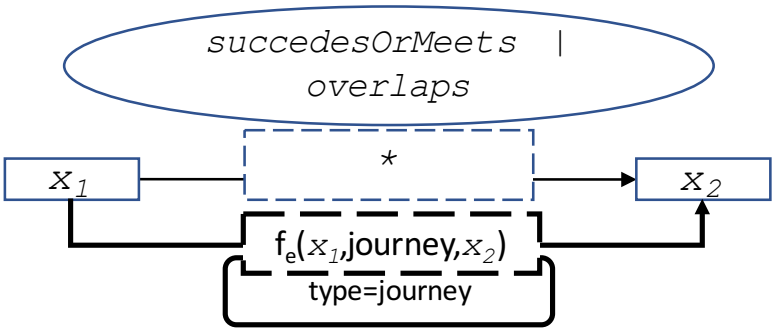


Figure 3.10: A journey pattern p_7.

3.3.4 Spread of Information

Question: For the event of interest, who are the top information leaders, i.e., those that started the spread of information and achieved the biggest exposure?

1. Select only the portion of the graph that relates to the topic of interest using the subgraph operator and a pattern with topic = 'Hurricane Sandy' pattern (not depicted):

$$\mathcal{G}_1 = \mathsf{subgraph}^T(p_6, wikitalk)$$

2. Connect every pair of nodes that has a path between them with non-decreasing time periods using the edge creation operator using pattern p_7 in Figure 3.10:

$$\mathcal{G}_2 = \mathsf{edge}^T_{p_7}(\mathcal{G}_1, \mathcal{G}_1)$$

3. Select the node with the largest number of edges using the vertex-subgraph operator and a pattern with max(sum(x)) expression (not depicted):

$$\mathcal{G}_3 = \mathsf{subgraph}^T(p_8, \mathcal{G}_2)$$

3.4 Formal TGA properties

Several properties of temporal languages have been studied: temporal groupedness, temporal semi-completeness, and temporal completeness [15]. We show that the TGraph model is temporally ungrouped but strongly equivalent to the canonical temporally grouped model. We also show that TGA is is snapshot reducible with respect to the nontemporal graph query language in Section 2.2.2.

Additionally, we show that TGA is not temporally semi-complete or temporally complete. We review each of these points in turn. Finally, we study the expressive power of a fragment of TGA.

3.4.1 Temporal Groupedness

Clifford et al. define two basic strategies for adding temporal information into the relational model: temporally ungrouped (tuple timestamping or first-normal-form) and temporally grouped (attribute timestamping or non-first-normal-form) [29]. Temporal relations in TRA, based on the TSQL2 model, are temporally ungrouped [15], since each tuple is associated with its period of validity. In such a model, a change to one attribute results in a new tuple. In a temporally grouped model related facts are grouped and their attributes are associated with their individual periods of validity with the use of a function.

Formally:

Definition 3.4.1 (Canonical temporally ungrouped relation model)**.** *[adapted from [29], pp. 69-70] Let $U_D = \{D_1, \ldots, D_{n_d}\}$ be a set of non-empty value domains, and $\mathbf{D} = \bigcup_{l=1}^{N_d} D_l$ be the set of all values. Let $\Omega^T = \{t_0, \ldots, t_i, \ldots\}$ be a non-empty, at most countably infinite, set of times with total order. Let $U_A = \{A_1, \ldots, A_n\}$ be a set of nontemporal attributes, and T a distinguished time attribute not in U_A.*

A temporally ungrouped (TU) relation schema R_{TU} is a 3-tuple $R_{TU} =< \mathbf{A}, \mathbf{K}, \mathbf{DOM} >$, where:

- $\mathbf{A} \cup \{T\} (\mathbf{A} \subseteq U_A)$ *is the set of attributes of the schema.*
- $\mathbf{K} \cup \{T\} (\mathbf{K} \subseteq \mathbf{A})$ *is the key of the schema, i.e.,* $\mathbf{K} \cup \{T\} \rightarrow \mathbf{A}$.
- $\mathbf{DOM} : \mathbf{A} \cup \{T\} \rightarrow U_D \cup \{\Omega^T\}$ *is a function that assigns domains to attributes in* $\mathbf{A}$ *and* T *to* Ω^T.

A TU database schema DB_{TU} is a finite set of TU relation schemas. A tuple rt on schema R_{TU} is a function that associates a value for each attribute $A_i \in \mathbf{A}$ in domain $\mathbf{DOM}(A_i)$ and a value in T to Ω^T. A TU relation is a finite set of TU tuples satisfying the key constraint.

Observe that the temporal relational schema defined in Section 2.1.1 is isomorphic to a temporally ungrouped relation structure, by using intervals instead of time points. This is formally shown in [15].

A canonical temporally grouped relation maintains the notion of an object changing over time. Formally:

Definition 3.4.2 (Canonical temporally grouped relation model)**.** *[adapted from [29], pp. 70-71] Let U_D, $\mathbf{D}$, Ω^T, and U_A be defined as for the canonical temporally ungrouped relation structure above. Any subset of $\mathbf{L} \subseteq \Omega^T$ is called a* lifespan*.*

A temporally grouped (TG) relation schema R_{TG} is a 3-tuple $R_{TG} =< \mathbf{A}, \mathbf{K}, \mathbf{DOM} >$, where:

- $\mathbf{A} \subseteq U_A$ *is the set of attributes of the schema.*
- $\mathbf{K} \subseteq \mathbf{A}$ *is the key of the schema, i.e.,* $\mathbf{K} \rightarrow \mathbf{A}$*.*
- $\mathbf{DOM} : \mathbf{A} \rightarrow U_D \cup \{\Omega^T\}$ *is a function that assigns to each attribute a value domain and the corresponding temporal domain.*

A TG database schema DB_{TG} is a finite set of TG relation schemas. A tuple rt on schema R_{TG} is a function that associates with each attribute $A_i \in \mathbf{A}$ a temporal function from the tuple lifespan to the domain assigned to the attribute $\mathbf{DOM}(A_i)$*. A TG relation is a finite set of TG tuples such that no two tuples agree on all keys at any instant in time.*

What makes TU relations ungrouped is that there is no mechanism to identify the tuples that correspond to the same entity over time, i.e., there is no unique grouped relation for an ungrouped relation. Without a grouping mechanism the TU model and the TG model are only *weakly* equivalent [29].

The TGraph model is ungrouped by the virtue of being expressed over an ungrouped relational model of TRA. However, the TGraph model requires that each node and edge have an id that serves as a key and persists over time, or, in Clifford terminology, it uses constant keys for a group identifier [29]. With this restriction the TGA model is strongly equivalent to the canonical temporally grouped model.

The reasons that we chose the ungrouped model are two-fold: a) it is the de-facto standard in the temporal relational database community after more than two decades of discussion, and b) it is easier to express temporal predicates over the ungrouped model.

3.4.2 Temporal Completeness

Böhlen, et al. define two kinds of upwards compatability with respect to a nontemporal model: temporal semi-completeness and temporal completeness [15]. To define both we need to recollect the notion of snapshot reducibility (Definition 2.1.4).

Snapshot reducibility states that for every nontemporal query q in language L, there must exist a corresponding temporal query q^t in the temporal language L^t that generalizes q. Note that this

definition does not pose any restrictions on the syntax of the temporal query, so it may be expressed quite differently than the nontemporal one. It also does not restrict the temporal language from having other operators with no nontemporal counterparts.

We can show that TGA is snapshot reducible with respect to a particular graph query language.

Theorem 2. *TGA satisfies the snapshot reducibility properties below with respect to the graph query language in Section 2.2.2.*

1. $\forall p \in \Omega^T(\mathit{snap}_p(\mathit{map}_v^T(f_v, \mathcal{G})) \Leftrightarrow \mathit{map}_v(f_v, \mathit{snap}_p(\mathcal{G})))$
2. $\forall p \in \Omega^T(\mathit{snap}_p(\mathit{map}_e^T(f_e, \mathcal{G})) \Leftrightarrow \mathit{map}_e(f_e, \mathit{snap}_p(\mathcal{G})))$
3. $\forall p \in \Omega^T(\mathit{snap}_p(\mathit{subgraph}^T(P, \mathcal{G})) \Leftrightarrow \mathit{subgraph}(P, \mathit{snap}_p(\mathcal{G})))$
4. $\forall p \in \Omega^T(\mathit{snap}_p(\mathit{agg}^T(P, \mathcal{G})) \Leftrightarrow \mathit{agg}(P, \mathit{snap}_p(\mathcal{G})))$
5. $\forall p \in \Omega^T(\mathit{snap}_p(\mathit{node}_a^T(P, \mathcal{G})) \Leftrightarrow \mathit{node}(P, \mathit{snap}_p(\mathcal{G})))$
6. $\forall p \in \Omega^T(\mathit{snap}_p(\mathit{edge}^T_{P=f_e(\mathcal{G}_1.v,\mathcal{G}_2.v),f_{v1}(k_1)=f_{v2}(k2)=\ldots=f_{vn}(k_n)=\mathit{any}}(\mathcal{G}_1, \mathcal{G}_2)) \Leftrightarrow \mathit{snap}_p(\mathcal{G}_1) + \mathit{snap}_p(\mathcal{G}_2))$ *(+ stands for graph join, per Def. 2.2.11)*
7. $\forall p \in \Omega^T(\mathit{snap}_p(\mathit{subgraph}^T(P_1, \mathit{edge}^T_{P_2,f_{v1}(k_1)=\ldots=f_{vn}(k_n)=\mathit{any}}(\mathcal{G}_1, \mathcal{G}_2))) \Leftrightarrow \mathit{snap}_p(\mathcal{G}_1) \odot_{\sigma 1,\sigma 2,g} \mathit{snap}_p(\mathcal{G}_2))$

The equivalences hold for arbitrary TGraphs*, except equivalence 6, graph join, which is defined specifically for disjoint graphs. The only restriction in this theorem is that any pattern P and mapping function f_v/f_e do not refer to the time attribute.*

Proof. To prove this Theorem we consider each equivalence in turn, using either the property graph (Def. 3.1.1) or the relational definition (Def. 3.1.3) as convenient.

- **vertex-map:**

  $\mathsf{snap}_p(\mathsf{map}_v^T(f_v, \mathcal{G})) = \mathsf{snap}_p(\mathsf{map}_v^T(f_v, (\mathsf{TV}, \mathsf{TE}))) = \mathsf{snap}_p(\ (\{(v, a'|T) \mid (v, a|T) \in \mathsf{TV} \wedge a' = f_v(v, a|T)\}, \mathsf{TE})\ ) = (\{(v, a') \mid p \in T \wedge (v, a|T) \in \mathsf{TV} \wedge a' = f_v(v, a|T)\}, \tau_p(\mathsf{TE}))$

  $\mathsf{map}_v(f_v, \mathsf{snap}_p(\mathcal{G})) = \mathsf{map}_v(f_v, \mathsf{snap}_p(\ (\mathsf{TV}, \mathsf{TE})\ )) = \mathsf{map}_v(f_v, (\tau_p(\mathsf{TV}), \tau_p(\mathsf{TE}))) = \mathsf{map}_v(f_v, (\{(v, a) \mid (v, a|T) \in \mathsf{TV} \wedge p \in T\}, \tau_p(\mathsf{TE}))) = (\{(v, a') \mid (v, a|T) \in \mathsf{TV} \wedge p \in T \wedge a' = f_v(v, a)\}, \tau_p(\mathsf{TE}))$

  To show that these two are equivalent, we exploit the commutativity of conjunction to rewrite the $p \in T \wedge (v, a|T) \in \mathsf{TV}$ to $(v, a|T) \in \mathsf{TV} \wedge p \in T$. Since the theorem restricts f_v from using time, $f_v(v, a|T) = f_v(v, a)$. The equivalence follows.

- **edge-map:**

$\mathsf{snap}_p(\mathsf{map}_e^T(f_e, \mathcal{G}))$ =

$\mathsf{snap}_p(\mathsf{map}_e^T(f_e, (\mathsf{TV}, \mathsf{TE}))) = \mathsf{snap}_p(\ (\mathsf{TV}, \{(e, v1, v2, a'|T) \mid (e, v1, v2, , a|T) \in \mathsf{TE} \wedge a' = f_e(e, v1, v2, , a|T)\})\ ) = (\tau_p(\mathsf{TV}), \{(e, v1, v2, a') \mid p \in T \wedge (e, v1, v2, a|T) \in \mathsf{TE} \wedge a' = f_e(e, v1, v2, a|T)\}, \tau_p(\mathsf{TE}))$

$\mathsf{map}_e(f_e, \mathsf{snap}_p(\mathcal{G})) = \mathsf{map}_e(f_e, \mathsf{snap}_p(\ (\mathsf{TV}, \mathsf{TE})\ )) = \mathsf{map}_e(f_e, (\tau_p(\mathsf{TV}), \tau_p(\mathsf{TE}))) = \mathsf{map}_e(f_e, (\tau_p(\mathsf{TV}), \{(e, v1, v2, a) \mid (e, v1, v2, a|T) \in \mathsf{TE} \wedge p \in T\})) = (\tau_p(\mathsf{TV}), \{(e, v1, v2, a') \mid (e, v1, v2, a|T) \in \mathsf{TE} \wedge p \in T \wedge a' = f_e(e, v1, v2, a)\})$

Just as with vertex-map, we exploit the commutativity of conjunction to rewrite the $p \in T \wedge (e, v1, v2, a|T) \in \mathsf{TE}$ to $(e, v1, v2, a|T) \in \mathsf{TE} \wedge p \in T$. Since the theorem restricts f_e from using time, $f_e(e, v1, v2, a|T) = f_e(e, v1, v2, a)$. The equivalence follows.

- **subgraph:**

$\mathsf{snap}_p(\mathsf{subgraph}^T(P, \mathcal{G})) = \mathsf{snap}_p(\ (q_v(P, \mathcal{G}), q_e(P, \mathcal{G}), \Pi, \rho, \xi^T, \lambda_v^T, \lambda_e^T))$

$\Downarrow$ apply definition of get-snapshot

$(\{v \mid v \in q_v(P, \mathcal{G}) \wedge \exists t(\xi^T(v, t) \wedge p \in t)\}, \{e \mid e \in q_e(P, \mathcal{G}) \wedge \exists t(\xi^T(e, t) \wedge p \in t)\}, \Pi, \rho, \lambda_v^T([p, p+1)), \lambda_e^T([p, p+1)))$

$\mathsf{subgraph}(P, \mathsf{snap}_p(\mathcal{G})) = \mathsf{subgraph}(P, (\{v \mid v \in V \wedge \exists t(\xi^T(v, t) \wedge p \in t)\}, \{e \mid e \in E \wedge \exists t(\xi^T(e, t) \wedge p \in t)\}, \Pi, \rho, \lambda_v^T([p, p+1)), \lambda_e^T([p, p+1))))$

$\Downarrow$ apply definition of subgraph

$(\{v \mid v \in V \wedge \exists t(\xi^T(v, t) \wedge p \in t) \wedge v \in q_v(P, (\{v \mid v \in V \wedge \exists t(\xi^T(v, t) \wedge p \in t)\}, \{e \mid e \in E \wedge \exists t(\xi^T(e, t) \wedge p \in t)\}, \Pi, \rho, \lambda_v^T([p, p+1)), \lambda_e^T([p, p+1))))\},$
$\{e \mid e \in E \wedge \exists t(\xi^T(e, t) \wedge p \in t) \wedge e \in q_e(P, (\{v \mid v \in V \wedge \exists t(\xi^T(v, t) \wedge p \in t)\}, \{e \mid e \in E \wedge \exists t(\xi^T(e, t) \wedge p \in t)\}, \Pi, \rho, \lambda_v^T([p, p+1)), \lambda_e^T([p, p+1))))\},$
$\Pi, \rho, \lambda_v^T([p, p+1)), \lambda_e^T([p, p+1)))$

$\Downarrow$ substitute graph snapshot with G

$(\{v \mid v \in V \wedge \exists t(\xi^T(v, t) \wedge p \in t) \wedge v \in q_v(P, G)\}, \{e \mid e \in E \wedge \exists t(\xi^T(e, t) \wedge p \in t) \wedge e \in q_e(P, G)\}, \Pi, \rho, \lambda_v^T([p, p+1)), \lambda_e^T([p, p+1)))$

The difference between the two expansions is only in the application of the q_v and q_e to a TGraph or a snapshot graph G. Per Def. 3.2.6, when the temporal navigational pattern P is restricted to have no temporal predicates, it is equivalent to a regular navigational graph pattern over each snapshot. Thus the snapshot reducibility property is observed by design.

- **aggregation:**

The proof for the aggregation operation is essentially the same as for the subgraph, since the aggregation operator uses a TNGP P^T, which reduces to a nontemporal pattern P when no temporal predicates are present.

- **node creation:**

  The proof for the node creation operation is essentially the same as for the subgraph, since the node creation operator uses a TNGP P^T, which reduces to a nontemporal pattern P when no temporal predicates are present.

- **graph join:**

  Edge creation is a more general operation than a graph join. Graph join is defined over disjoint graphs, meaning that no property resolution is needed for nodes or edges. The graph pattern is equivalent to the following conjunctive query: $\sigma^T_{f_e(v_1,v_2),v1,v2}(TV_1 \bowtie^T TV_2)$.

  $\mathsf{snap}_p(\mathsf{edge}^T_{P==f_e(\mathcal{G}_1.v,\mathcal{G}_2.v),f_{v1}(k_1)=f_{v2}(k2)=\ldots=f_{vn}(k_n)=\mathsf{any}}(\mathcal{G}_1,\mathcal{G}_2)) =$
  $\mathsf{snap}_p((\mathcal{R}(f_{v1},\ldots,f_{vn},TV_1 \cup^T TV_2)),q_e(P,\mathcal{G}_1,\mathcal{G}_2)) = \mathsf{snap}_p((TV_1 \cup^T TV_2,q_e(P,\mathcal{G}_1,\mathcal{G}_2))) =$
  $(\tau_p(TV_1 \cup^T TV_2),\tau_p(q^T_e(P,\mathcal{G}_1,\mathcal{G}_2))) = (\tau_p(TV_1) \cup \tau_p(TV_2),q_e(P,\mathcal{G}_1,\mathcal{G}_2))$

  We take advantage of the fact that (a) the resolve operation is a no-op for a disjoin graph and that the temporal union operator is snapshot reducible to the snapshot union operator.

  $\mathsf{snap}_p(\mathcal{G}_1) + \mathsf{snap}_p(\mathcal{G}_2) = (\tau_p(TV_1),\tau_p(TE_1)) + (\tau_p(TV_2),\tau_p(TE_2)) = (\tau_p(TV_1) \cup$
  $\tau_p(TV_2),\tau_p(TE_1) \cup \tau_p(TE_2) \cup \{\forall v_1 \in \tau_p(TV_1) \forall v_2 \in \tau_p(TV_2) f_e(v_1,v_2)\})$

  The only difference between the two expansions is regarding the pattern application. It has been proven in [16] that a temporal join is snapshot reducible to the nontemporal join. Thus the equivalence holds.

- **graph composition:**

  The graph composition equivalence proof is the same as the graph join, but with a different pattern that creates a new edge for every pair of edges in the correct direction.

□

As noted, snapshot reducibility places no restrictions on the syntax of the temporal language. To deal with the syntax, Böhlen, et al., define a more restrictive type of upward compatability termed *temporal semi-completeness* [15] or S-reducibility [16].

Definition 3.4.3 (Temporal semi-completeness)**.** *[[15], p. 162] Let $M = (DS, L)$ be a nontemporal data model and $M^T = (DS^t, L^t)$ be a valid-time temporal data model. Then data model M^t is temporally semi-complete with respect to M iff:*

- *For every relation $\mathbf{r}$ in DS there exists a temporal relation $\mathbf{r^t}$ in DS^T such that $\mathbf{r} = \tau_t(\mathbf{r^t})$.*
- *For every query q in L, there exists a query q^t in L^t that is snapshot reducible with respect to q.*
- *There exist two text strings S_1 and S_2 such that for all pairs of queries (q, q^t), where q^t is snapshot reducible to q, query q^t is syntactically identical to $S_1 q S_2$.*

The intent of temporal semi-completeness is to provide a kind of upward compatability that simplifies the transition of nontemporal language users to the temporal variant. For example, TSQL2 provides a VALID keyword that indicates that the query is over the valid-time temporal semantics. Interestingly, it has been shown that TSQL2 is not temporally semi-complete with respect to SQL-92 because of a) its restriction to duplicate free set semantics, and b) lack of snapshot reducible counterparts for nested queries [15].

The TGraph model meets condition 1, since every snapshot graph can be modeled as a TGraph with a unit interval as its timestamp. The use of duplicate free set-based semantics is not an issue because graph snapshots do not allow duplicates either – every node and edge has an identity. Condition 2 is also satisfied, as we already showed that for every graph operator there is a corresponding TGA operator that is snapshot-reducible.

Since TGA does not specify a syntax, requirement 3 cannot be satisfied. The definitions in Section 3.2 specify the semantics of each operator, but do not dictate any specific way the operator must be expressed by the user. For instance, we do not specify how the patterns are specified. Thus TGA cannot be considered temporally semi-complete with respect to any graph query language.

A further notion of upward compatability is *temporal completeness.*

Definition 3.4.4 (Temporal completeness)**.** *[[15], p. 166] Let $M = (DS, L)$ be a nontemporal data model and $M^T = (DS^t, L^t)$ be a valid-time temporal data model. Then data model M^t is temporally complete with respect to M iff:*

- *M^t is temporally semi-complete with respect to M.*
- *It is possible to override snapshot reducibility for every snapshot reducible query q^t in L^t by some syntactic means such as dropping the syntactic extensions that enforce it.*

- *Syntactic reducibility can be applied to subqueries.*
- *Allen's interval operators [5] or their equivalents can be used in queries.*
- *It is possible to refer to both the (a) coalesced interval and (b) original intervals as specified by the user.*

Condition 2 is elsewhere termed *nonrestrictiveness* [16] and gives the user full control over the timestamps as regular values if desired. Condition 5 essentially refers to the change preservation property of sequence semantics [35], without which the intervals specified by the user have no special meaning. Any point semantics coalesces value-equivalent consecutive and overlapping tuples and does not preserve user-specified intervals.

TGA is not temporally complete both because it is not temporally semi-complete and because it uses point semantics, thus violating the final condition.

3.4.3 Expressive Power

In this section we study expressiveness of the TGraph model, which consists of the TGraph data structure (Definition 3.1.3) and of TGA, an algebra for querying the data structure (Section 3.2). We stress that ours is a valid-time data model that does not provide transaction-time and bi-temporal support.

Important note: We restrict our attention to a subset of TGA operations, excluding recursive queries. That is, every operation where Temporal Navigational Graph Patterns (Def. 3.2.6) are applied is restricted to Basic Graph Patterns (Def. 2.2.2) with temporal predicates. A Basic Graph Pattern is equivalent to a regular conjunctive query without projection [3]. We also restrict the property values to atomic types for the purposes of this discussion.

We start by proposing two natural notions of completeness for a temporal graph query language.

Definition 3.4.5. *Let L^t be a temporal relational language and $\mathcal{G}$ — a relational representation of a temporal graph. An edge-query q_e^t in L takes a graph $\mathcal{G}(\mathsf{TV}, \mathsf{TE})$ as input, and outputs another graph $\mathcal{G}'$ on the vertices of $\mathcal{G}$ such that the edges of $\mathcal{G}'$ are defined by q_e^t. A language is L^t-edge-complete if it can express each q_e^t in L^t.*

Note that the query q_e^t is not restricted to act on TE alone, and may refer to the other constituent relation of $\mathcal{G}$, namely, TV.

Definition 3.4.6. *Let L^t be a temporal relational language, and let $\mathcal{G}$ be a relational representation of a temporal graph. A vertex-query q_v^t in L^t takes a graph $\mathcal{G}(\mathsf{TV}, \mathsf{TE})$ as input, and outputs another*

graph $\mathcal{G}'$ such that the vertices of $\mathcal{G}'$ are defined by q_v^t. A language is L^t-vertex-complete if it can express each q_v^t in L^t.

We now refer to definitions 3.4.5 and 3.4.6 and show that TGA is edge-complete and vertex-complete, with respect to the valid-time fragment of temporal relational algebra (TRA). TRA is an algebra that corresponds to temporal relational calculus [28], a first-order logic that extends relational calculus, supporting variables and quantifiers over both the data domain and time domain.

Theorem 3. *TGA is TRA-edge-complete.*

Proof. The result of every conjunctive edge-query over the vertices of $\mathcal{G}$ can be expressed by $\sigma_c(\mathsf{TV} \times^T \mathsf{TV})$. Queries of this kind can be expressed by the edge creation operator of TGA (Section 3.2.9), invoked as:
$\mathsf{edge}^T(q = \sigma_c(\mathcal{G}_1.\mathsf{TV} \times^T \mathcal{G}_2.\mathsf{TV}), \mathcal{G}_1 = \mathcal{G}, \mathcal{G}_2 = \mathcal{G})$ □

Theorem 4. *TGA is TRA-vertex-complete.*

Proof. Every TRA vertex-query can be expressed in TGA by a sequence of vertex-subgraph $q_v^T(\mathcal{G})$ (Section 3.2.3) and attribute-based node creation node_a^T (Section 3.2.5). Attribute-based node creation supports Skolem functions, and is necessary to handle queries that introduce vertex identifiers. □

Chapter 4: System

We developed a prototype system Portal that supports TGA operations in a distributed environemtn. All TGraph operations are available through the public API of the Portal library. In this Chapter we describe the system implementation, including its architecture (Section 4.2), physical data layout (Section 4.3), and various in-memory data layouts (Section 4.4). We conclude this chapter with a discussion about the limitations of the current prototype.

4.1 Background

Apache Spark[1] is a distributed open-source system similar to MapReduce [34], but based on an in-memory processing approach [95]. Data in Spark is represented by **Resilient Distributed Datasets (RDDs)**, a distributed memory abstraction for fault-tolerant computing. All operations on RDDs are treated as a series of transformations on a collection of data partitions, such that any lost partition may be recalculated based on its lineage.

Spark Scheduler builds a directed acyclic graph (DAG) of the RDD transformations and splits the DAG into stages of tasks (Figure 4.1). Tasks within the same stage are executed in parallel by the workers, as assigned by the cluster manager. A task processes one partition of the data and partitions are stored in worker memory. When there is insufficient memory, partitions may be cached to disk.

Spark RDD transformations are functional language operations that can be mapped to relational operators. For example, a filter operation is similar to relational selection. The transformations are lazy and are not executed until an *action* is requested, such as retrieving the results to print or saving to disk.

Spark is a large library that has RDDs at its core, but also includes SparkSQL [9], a GraphX library for graphs [44], as well as streaming and machine learning modules, among others. SparkSQL provides a SQL parser over Datasets, a relation abstraction on top of RDDs. GraphX provides a Graph abstraction and an API for graph transformations such as subgraph, map vertices/edges,

[1] http://spark.apache.org/

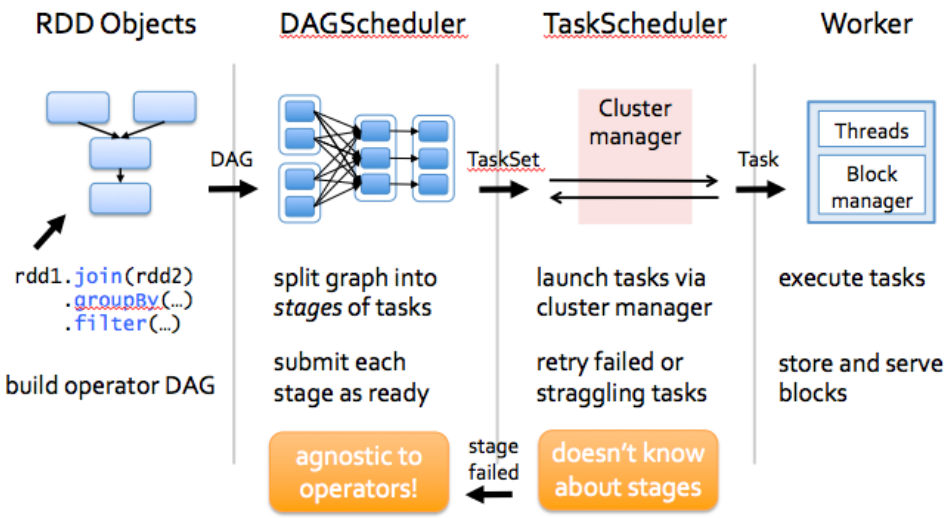


Figure 4.1: Apache Spark task processing architecture. ©Arush Kharbanda. [56].

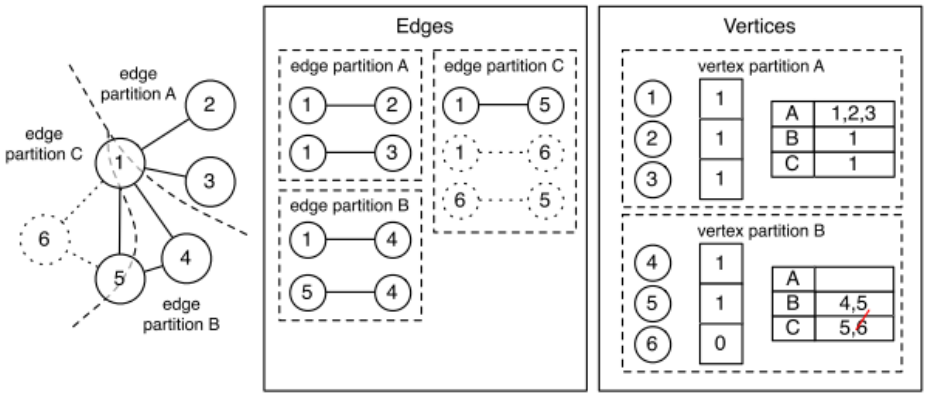


Figure 4.2: Distributed representation of a graph, Fig.3 in [44].

and a Pregel API. A Graph in GraphX is an RDD of edges and a special replicated RDD of nodes (Figure 4.2). In other words, GraphX uses a vertex-cut approach where each edge resides in a partition, and vertices are replicated to each partition where they are either a source or a destination node for an edge. Vertex-cut has been shown to improve performance of graph computations, and specifically iterative whole-graph computations such as PageRank, by minimizing communication in natural graphs that exhibit a powerlaw degree distribution [43].

Whole-graph iterative computation (analytics) is a special case of the aggregation graph operation (Def. 2.2.8). In a distributed environment a common approach for support of analytics is using the "Thinking like a Vertex" or Pregel abstraction [72]. The Pregel abstraction is a bulk synchronous execution model of whole-graph computation consisting of a vertex program, a message combiner, and a send message program, which was first introduced by Malewicz et al. in [71]. Because the computation is vertex-centric, it is also sometimes called Think Like a Vertex. See [72] for a survey of vertex-centric frameworks for large graphs.

Pregel computation is broken up into **supersteps** for the purposes of synchronization. At each superstep messages are issued by some nodes, are combined into a single message to reduce communication costs, and each vertex updates its state based on the messages it receives. At the end of the update each node can stay active or vote to halt. When all nodes have voted to halt, the computation is complete. It can also be stopped after a desired number of iterations.

Listing 4.1: Connected components computation in GraphX.

```
def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]):
   Graph[VertexId, ED] = {
  val ccGraph = graph.mapVertices { case (vid, _) => vid }
  def sendMessage(edge: EdgeTriplet[VertexId, ED]):
     Iterator[(VertexId, VertexId)] = {
    if (edge.srcAttr < edge.dstAttr) {
      Iterator((edge.dstId, edge.srcAttr))
    } else if (edge.srcAttr > edge.dstAttr) {
      Iterator((edge.srcId, edge.dstAttr))
    } else {
      Iterator.empty
    }
  }
  val initialMessage = Long.MaxValue
```

```
  Pregel(ccGraph, initialMessage, activeDirection = EdgeDirection.Either)(
    vprog = (id, attr, msg) => math.min(attr, msg),
    sendMsg = sendMessage,
    mergeMsg = (a, b) => math.min(a, b))
}
```

Listing 4.1 shows a Weakly Connected Components computation in Pregel in GraphX. The vertex program, vprog, is simply a minimum of all vertex ids seen by the vertex, including its own. The message combiner, mergeMsg, also only selects a minimum vertex id. The message generation happens at the edges, in the sendMessage method, which issues a new message to the edge endpoint with the smaller seen node id.

To make the Pregel computation efficient, it is important to partition the graph in such a way as to minimize the communication costs between the graph partitions. A good partitioning strategy needs to be balanced, assigning an approximately equal number of units to each partition, and limit the number of cuts across partitions, reducing cross-partition communication. GraphX provides several partition strategies. The E2D edge partitioning strategy uses a sparse edge adjacency matrix that is partitioned in two dimensions, guaranteeing a $2\sqrt{n}$ bound on vertex replication, where n is the number of partitions. E2D has been shown to provide good performance for Pregel-style analytics [44].

In addition to the SparkSQL and GraphX modules that come packaged with Spark, we also use the Spark Datalog third-party module [82]. Spark Datalog accepts String queries expressed in Datalog over SparkSQL relations and accepts monotonic aggregates.

4.2 Architecture

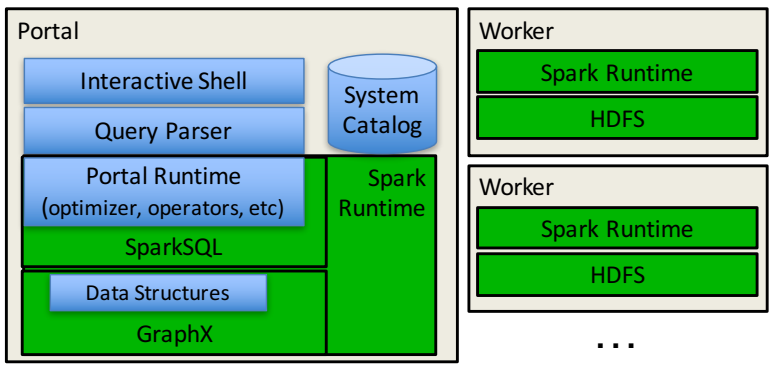


Figure 4.3: Portal system architecture.

Figure 4.3 shows the Portal system architecture on top of Apache Spark. We selected Apache Spark because it is a popular open-source system, because of its in-memory processing approach, and because of a wide range of support modules available in it, e.g., SparkSql, Datalog, GraphX. Green boxes indicate built-in Spark components, while blue are those we added for Portal. The Portal system includes a TGraph API, several in-memory representations, and several optimizations. The data is distributed in partitions across the cluster workers, read in from Hadoop Distributed File System (HDFS)[2], and can be viewed both as a graph and as a pair of Spark Resilient Distributed Datasets (RDDs). The API includes all operations of the TGA as well as point local methods (get node/edge at time t, get snapshot at time t, get node/edge history). We include the point local methods for convenience, since they are outside of the TGA. The Portal API exposes node/edge RDDs to the user and provides convenience methods to convert them to Spark Datasets. Arbitrary SparkSQL queries can then be executed over these relations.

Listing 4.2: Portal API.

```
def size(): Interval
def vertices: RDD[(VertexId,(Interval, VD))]
def edges: RDD[TEdge[ED]]
def getTemporalSequence: RDD[Interval]
def getSnapshot(time: LocalDate):Graph[VD,(EdgeId,ED)]
def coalesce(): TGraph[VD, ED]
def trim(bound: Interval): TGraph[VD, ED]
def vmap[VD2: ClassTag](map: (VertexId, Interval, VD) => VD2, defVal: VD2)
   (implicit eq: VD =:= VD2 = null): TGraph[VD2, ED]
def emap[ED2: ClassTag](map: TEdge[ED] => ED2): TGraph[VD, ED2]
def vsubgraph(pred: (VertexId, VD, Interval ) => Boolean ): TGraph[VD,ED]
def esubgraph(pred: TEdgeTriplet[VD,ED] => Boolean,
   tripletFields: TripletFields = TripletFields.All): TGraph[VD,ED]
def aggregateMessages[A: ClassTag]
   (sendMsg: TEdgeTriplet[VD,ED] => Iterator[(VertexId, A)],
    mergeMsg: (A, A) => A, defVal: A,
    tripletFields: TripletFields = TripletFields.All): TGraph[(VD, A), ED]
def createTemporalNodes(window: WindowSpecification, vquant: Quantification,
   equant: Quantification, vAggFunc: (VD, VD) => VD,
   eAggFunc: (ED, ED) => ED): TGraph[VD, ED]
```

[2] https://hadoop.apache.org/

```
def createAttributeNodes( vAggFunc: (VD, VD) => VD)
   (vgroupby: (VertexId, VD) => VertexId ): TGraph[VD, ED]
def union(other: TGraphNoSchema[VD, ED], vFunc: (VD, VD) => VD ,
   eFunc: (ED, ED) => ED): TGraphNoSchema[VD, ED]
def difference(other: TGraphNoSchema[VD, ED]): TGraphNoSchema[VD, ED]
def intersection(other: TGraphNoSchema[VD, ED], vFunc: (VD, VD) => VD,
   eFunc: (ED, ED) => ED): TGraphNoSchema[VD, ED]
def pregel[A: ClassTag]
   (initialMsg: A, defaultValue: A, maxIterations: Int = Int.MaxValue,
     activeDirection: EdgeDirection = EdgeDirection.Either)
   (vprog: (VertexId, VD, A) => VD,
     sendMsg: EdgeTriplet[VD, (EdgeId,ED)] => Iterator[(VertexId, A)],
     mergeMsg: (A, A) => A): TGraph[VD, ED]
```

The TGraph API is show in Listing 4.2.

4.3 Physical Layout

Our on-disk data layout uses Apache Parquet [3], a columnar data format for HDFS based on the Dremel project [73]. Nodes are stored in node files and edges in edge files, consistently with the GraphX API.

4.3.1 Locality

Following the approach in ImmortalGraph [74], we investigated two different types of on-disk locality: structural and temporal. In ImmortalGraph, the structural locality layout places all the evolution data for a particular snapshot together, thus preserving the *structure* of a snapshot. The time locality places all the evolution data for a node together and provides a node index, such that no sequential scan of all the nodes is required.

We adapt the locality methods of ImmortalGraph for a distributed environment. Apache Parquet does not have a mechanism for indexing. However, it supports filter pushdown on any column, by which the data is sorted on disk. Filter pushdown is the only pseudo-indexing method that Parquet provides. However, Parquet is the de facto standard for distributed file systems based on the combination of read performance and compactness, and, as we show in Chapter 5, its performance in Portal is sufficient.

[3] https://parquet.apache.org/

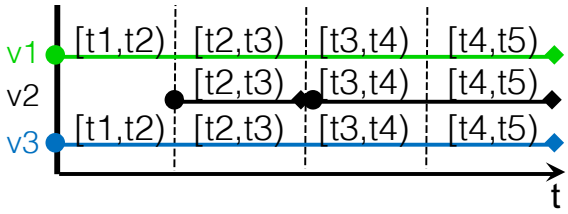


Figure 4.4: Tuple intervals are split into four snapshot groups.

Unlike ImmortalGraph, we store materialized node/edge tuples, rather than deltas. The node (resp. edge) files are sorted by the interval start time and node (resp. edge) id. Structural locality uses time as the first sort and id as the second, thus placing nodes/edges from the same time together. Temporal locality uses id as the first sort and time as the second, thus placing the whole history of a node/edge together. Unlike in ImmortalGraph, Portal can use both sorted columns for filter pushdown together. For example, to select only nodes with periods of validity in interval $[2010/01, 2011/01)$, we use the following predicate: NOT (estart >= 2011/01 OR eend <= 2010/01).

Temporal locality places the history of each node (resp. edge) in the same block, which improves loading time for all in-memory representations and reduces communication costs during coalescing. We present the effectiveness of both locality types on local and global operations in Section 5.2.

4.3.2 Snapshot Groups

In addition to sort order, datasets are partitioned into **snapshot groups**, a concept also introduced in the ImmortalGraph system [74]. A snapshot group represents the whole evolving graph history for one period and contains a node archive and an edge archive. The full history of the evolving graph is a sequence of nonoverlapping consecutive time intervals, one for each snapshot group. All the tuples with the periods of validity within the snapshot group interval belong to the group. A tuple with a period of validity that spans multiple snapshot groups is split and placed into each group it spans. This is shown in Figure 4.4, where three tuples are split into four snapshot groups, causing, for instance, tuple v1 to be split into 4 tuples.

Multiple snapshot group methods are feasible. We investigated five methods: no-partitioning, equi-width partitioning, equi-width by change, equi-depth partitioning, and redundancy partitioning.

No-Partitioning

No-partitioning method places all the tuples into a single snapshot group that covers the whole evolving graph history. This method is beneficial compared to all the others because the data is

Table 4.1: A co-authorship network from Table 3.1, reproduced here for convenience.

TV

v	**a**	**T**
1	type=person,name=Alice,school=Drexel	[2015/1,2015/7)
2	type=person,name=Bob	[2015/2,2015/5)
2	type=person,name=Bob,school=CMU	[2015/5,2015/10)
3	type=person,name=Cathy,school=Drexel	[2015/1,2015/10)

TE

e	**v1**	**v2**	**a**	**T**
1	1	2	type=co-author,cnt=3	[2015/2,2015/6)
2	2	3	type=co-author,cnt=4	[2015/7,2015/10)

coalesced. In contrast, once even a single tuple is split, the whole dataset is considered uncoalesced, which impacts performance of some operations. We discuss coalescing implementation and requirements in Section 4.5.

Consider simple TGraph depicted in Table 4.1. With no-partitioning method we compute a single node file spanning interval $[2015/1, 2015/10)$ and a single edge file with the same interval.

Equi-Width Partitioning

The period of the whole evolving graph history is split into a desired number of snapshot groups of equal calendar duration. I.e., if the dataset covers ten years and five groups are computed, each group will be two years long. This type of partitioning method is very easy to compute. However, most evolving graph datasets are heavily skewed, that is, they start small and grow in size with time. Thus the resulting snapshot groups are unbalanced.

Consider the same TGraph again (Table 4.1). With equi-width partitioning and a configuration input 3 months, we generate 3 node files for periods $[2015/1, 2015/4), [2015/4, 2015/7), [2015/7, 2015/10)$, and 3 edges files for the same periods.

Equi-Width by Change

This method is a variant of the equi-width partitioning. Instead of using a calendar to divide into groups, each group receives the same number of representative graphs. A representative graph is a single graph associated with a period of validity during which no changes to any of the graph nodes or edges, or their associated properties, occurred. In other words, a representative graph is

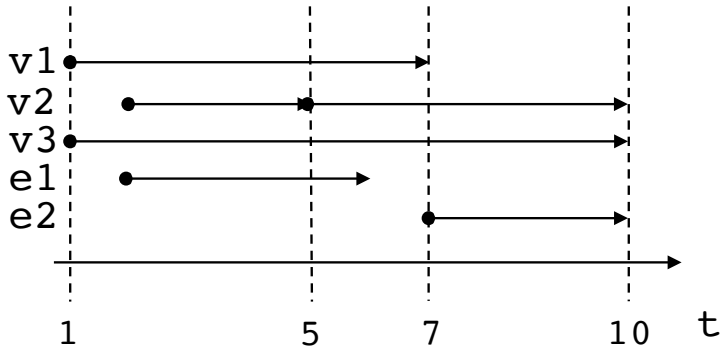


Figure 4.5: Tuples of TV and TE in TGraph from Table 4.1, split into three equi-depth groups.

a sequence of equivalent graph snapshots, to distinguish from a single snapshot that is associated with a single time instant.

TGraph in Table 4.1 consists of five representative graphs. Equi-width by change partitioning with a configuration input 3 graphs generates 2 node files for periods $[2015/1, 2015/6), [2015/6, 2015/10)$. If TGraph contains some change every month at one-month resolutions, then the same periods would be computed as by the regular equi-width partitining.

Equi-Depth Partitioning

The equi-depth partitioning produces a desired number of snapshot groups such that the maximum group size is minimized. The group size is the number of node and edge tuples that the group contains. This method is based on the optimal splitters work by Le et al. [64], adapted for graphs. The objective is to partition the evolving graph among machines in a cluster to achieve load balancing. For load balancing we want to place approximately equal number of intervals in each partition. This cost model is based on the assumption that the target cluster is homogeneous and time locality is required. For example, temporal aggregation γ^T requires such locality to compute aggregates at each time instant and its performance is a function of the size of the partition.

To understand how equi-depth partitioning works, we need to visualize TGraph in Table 4.1 as a bag of intervals on a timeline. Figure 4.5 shows the intervals, with the dashed lines indicating snapshot group boundaries if we partition into three groups. Year is not depicted for clarity, since it is the same in all tuples. The three snapshot group intervals are $[2015/1, 2015/5), [2015/5, 2015/7), [2015/7, 2015/10)$. Clearly these intervals are not of the same duration. However, each group contains at most four intervals, minimizing the number of interval cuts with balanced loading.

Redundancy Partitioning

This method divides the node and edge tuples into groups to minimize redundancy with the following group. The desired maximum redundancy threshold is used as an input parameter. Each time instant is included into the current group if the redundancy of the group is above the threshold. In practice the computation is made over representative graphs rather than all time instances. To account for the data skew, the redundancy between any two snapshots is normalized according to the following formula:

$2 * (|V_1 \cap V_2| + |E_1 \cap E_2|)/(|V_1 \cup V_2| + |E_1 \cup E_2|)$,

where V_1 is the set of node tuples from the previos snapshot (resp. E_1 of edges) and V_2 is the set of node tuples from the current snapshot (resp. E_2 of edges). The maximum redundancy ratio is 1.0.

Because some intervals are split, snapshot groups produce uncoalesced TGraphs. Care must be taken when selecting the snapshot group method for a particular operation, based on whether the operation can be caried out correctly over the uncoalesced input.

Based on an evaluation (see Section 5.2) we selected the no-partitining and the depth partitioning methods to create snapshot groups. A command-line option can be used to select the snapshot group method to use, to support experimental comparison between the methods. For queries that require coalesced data, such as subgraph, no-partitioning method is recommended.

4.4 In-Memory Representations

Portal provides four in-memory TGraph representations that differ in compactness and the kind of locality they prioritize. VertexEdge (VE) is a direct translation of the model and the most compact representation. RepresentativeGraph (RG) stores each representative graph explicitly, and so naturally preserves structural locality, but temporal locality is lost. OneGraph (OG) stores all nodes and edges of an evolving graph once, in a single data structure. This representation emphasizes temporal locality. HybridGraph (HG) trades compactness for better structural locality, by aggregating together several consecutive snapshots, and computing a OneGraph for each group. We can convert from one representation to any other at a small cost, so it is useful to think of them as access methods in the context of individual operations.

Each in-memory representation loads the data from the same on-disk layout with structual locality, but using different loading techniques, as described below.

To maintain interoperability with GraphX, we use the long datatype to represent node and edge ids. This gives a sufficient range for all large datasets currently available. However, this design decision, when combined with the distributed nature of the system, has an implication on the implementation of Skolem functions for node and edge creation (Defs. 3.2.9 and 3.2.14 respectively). Specifically, each machine in the cluster must be able to assign ids to new nodes or edges such that the id has not been already used and that the assignment is done consistently across all machines. By a consistent assignment of ids we mean that, e.g., a new node for school Drexel must be assigned the same unique id by any cluster machine that computes a tuple for it. In the current prototype, the Skolem function is a user-provided function. This shifts the responsibility to guarantee consistent assignment to the user. In our experiments we used a simple id assignment based on a hashing function. This is an obvious limitation that we discuss in additional detail in Section 4.6.

Table 4.2 summarizes how each data structure supports each TGA operator. We now describe each in-memory representation in detail and elaborate on the summary of Table 4.2.

4.4.1 VertexEdge

VertexEdge (VE) is a direct implementation of the TGraph-relational model (Def. 3.1.3), and is the most compact when there is only a single property: one RDD contains all nodes and another all edges. Consistently with the model and the GraphX API, all node properties are stored together as a single nested attribute, as are all edge properties. The main advantage of this schema-less attribute representation is that it can easily deal with schema evolution and leaves the details of attribute processing to the user. The disadvantage is that different properties may have different evolution rates, and a change to a single property requires a new attribute tuple.

Both the node and edge RDDs are loaded from the parquet files and no additional transformations are required.

VE implements the TGA operators through a sequence of RDD transformations roughly equivalent to nontemporal relational algebra operators. Apache Spark is not a temporal DBMS but rather an open-source in-memory distributed framework that combines graph parallel and data parallel abstractions. Following the approach of Dignos et al. [35] we rewrite our temporal operators into a sequence of nontemporal relational operators or their equivalents for Spark RDDs, maintaining point semantics. This allows our algebra to be implemented in any nontemporal relational database. In total, we need the **resolve** primitive we introduced in Definition 3.2.2, two primitives

Table 4.2: TGA operators in each in-memory representation.

Operation	**VE**	**RG**	**OG**	**HG**
trim	filter V and E; modify periods to be within trim interval	slice sequence of RGs	filter indices in bitsets	slice sequence of OGs and filter indices in remaining OGs
vertex- and edge-map	apply map function to each element; coalesce	apply map to each element within each RG; coalesce	N/A	N/A
vertex- and edge-subgraph (basic pattern)	filter V ad E; enforce FK constraint	only when predicate not on interval: filter V and E of each RG	N/A	N/A
vertex- and edge-subgraph (recursive pattern)	Spark Datalog integration	Spark Datalog integration	Spark Datalog integration	Spark Datalog integration
aggregation (non-recursive pattern only over snapshots)	compute edge triplets; reduce by key; left outer join with previous node values	use built-in GraphX aggregateMessages method; left outer join with previous node values	use built-in GraphX aggregateMessages method; left outer join with previous node values	like in OG

Table 4.2: TGA operators in each in-memory representation, continued.

aggregation (snapshot analytics)	N/A	compute for each RG	compute for all periods simultaneously	compute for all periods within each OG simultaneously
attribute-based node creation	map each node to new id; join edges with new nodes to get new id; rest as above	within each window map nodees and edge triplets to new ids; rest as above	N/A	N/A
temporal-window node creation	split each node/edge by window; reduce by window key; filter those under quantification threshold; enforce FK constraint	map each RG to windows; group nodes/edges in each window into one RG, filtering by quantification	only structure: for each node/edge map indices in bitsets to corresponding windows; filter by quantification threshold; enforce FK constraint	only structure: combine OGs as necessary to group into windows; map indices in bitsets; filter by quantification threshold; enforce FK constraint
union	compute combined intervals; split each node/edge by new intervals; full outer join	compute combined intervals; combine RGs in corresponding intervals; full outer join	structure only: remap bitsets to new intervals; union nodes/edges from two graphs and reduce by key to combine bitsets	structure only: combine OGs as necessary; rest as in OG
intersection	compute intervals; split each node/edge by new intervals; inner join	compute intervals; inner join of nodes/edges from corresponding intervals	structure only: like union but with bitset intersection	structure only: like union but with bitset intersection

Table 4.2: TGA operators in each in-memory representation, continued.

difference	compute intervals; split each node/edge by new intervals; left outer join and filter	compute combined intervals; outer join of nodes for each pair of RGs in corresponding intervals; filter	structure only: compute combined intervals; remap bitsets to new intervals; outer join of nodes; filter	structure only: combine OGs as necessary, rest as in OG
edge creation	Spark Datalog integration	Spark Datalog integration	Spark Datalog integration	Spark Datalog integration

to maintain TRA model (**coalesce** and **constrain**, which enforces foreign key constraint from TE to TV), and the primitives described in [35] to translate TRA operators into relational operators: **extend** and **normalize**. Because our model uses point semantics and does not require change preservation, we do not need the **align** primitive of [35] and can use the **normalize** primitive in its place.

The **coalesce** primitive merges adjacent and overlapping time periods for value-equivalent tuples. This operation, which is similar to duplicate elimination in conventional databases, has been extensively studied in the literature [13, 98]. Several implementations are possible for the coalesce operation over temporal SQL relations. Because Spark is an in-memory processing system, we use the partitioning method, where the relation is grouped by key, and tuples are sorted and folded within each group to produce time periods of maximum length. Eager coalescing, however, is not desirable since it is expensive and some operations may produce correct results (up to coalescing) *even when computing over uncoalesced inputs.* We describe an optimization based on lazy coalescing in Section 4.5.

The **resolve** primitive is implemented using a group by key operation in Spark and convenience methods on the property set class. The property set class supports adding all properties from another set such that they are combined by key, and applying aggregation functions one at a time for each property name.

The **constrain** primitive constrains one relation with respect to another, such as removing edges from the result that do not have associated nodes, or trimming the edge validity period to be within the validity periods of associated nodes. It is introduced here because Spark does not have a built-in way to express foreign key constraints. We do this by executing a join of the two relations — either a broadcast join or a hash join — and then adjusting time periods as necessary. This is an expensive operation and is only performed when necessary as determined by the soundness analysis, e.g., when vertex-subgraph has a non-trivial predicate over $\mathcal{G}$, and when node creation has a more restrictive node quantifier r_v than edge quantifier r_e. This optimization is described in detail in Section 4.5

The **split** primitive maps each tuple in relation R into one or more tuples based on a temporal window expression such as w=3 months. This is done in place of the more computationally expensive $\bowtie^T W$ operation in the temporal-window node creation operator. A purely relational implementation of this primitive is possible with the use of a special Chron relation that stores all possible time points of the temporal universe and supports computation without materialization.

Another approach is to introduce fold and unfold functions that can split each interval into all its constituent time points. Both of these approaches have strong efficiency concerns, see [14] for an in-depth discussion. In Spark we are not limited to relational operators only and can use functional programming constructs. **Split** can be efficiently implemented with a flatMap, which emits multiple tuples as necessary by applying a lambda function and flattening the result. We use this method in our implementation.

The **extend** primitive extends a relation with an additional attribute that represents the tuple's timestamp, see [35] for a definition. Extend allows explicit references to timestamps in operations, and is needed for extended snapshot reducibility. We implement extend by defining an Interval class and including it as a field in every RDD.

The **normalize** primitive produces a set of tuples for each tuple in $\mathbf{r}$ by splitting its timestamp into non-overlapping periods with respect to another relation $\mathbf{s}$ and attributes $\mathbf{B}$. See [35] for the formal definition. Intuitively, normalize creates tuples in corresponding groups such that their timestamps are also equivalent. For example, for a pair of relations $\mathbf{r}$ with single tuple $(v_1, [2016/05/01, 2016/08/01))$ and $\mathbf{s}$ with single tuple $(v_1, [2016/07/01, 2016/09/01))$, it splits the intervals to non-overlapping fragments. $\mathcal{N}_A(\mathbf{r}, \mathbf{s}) = (v_1, [2016/05/01, 2016/07/01)), (v_1, [2016/07/01, 2016/08/01))$. This primitive is necessary for node creation, set operators like union, and joins. The **normalize** primitive relies on an efficient implementation of the tuple splitter. We split each tuple based on the change periods over the whole graph, avoiding costly joins but potentially splitting some tuples unnecessarily.

VE supports all TGA operations in some form. Currently the subgraph operation is separated into the vertex- and edge-subgraph operators and supported by integration with Spark Datalog [82], as is the edge creation. We ported Spark Datalog to the latest version of Spark[4] and extended it with temporal predicates (overlaps, datediff, intersection) so that computations may be expressed over the TV and TE relations.

The attribute-based node creation is limited to the property values available within individual nodes and returns a new graph of only the new nodes. The aggregation operator accepts a non-recursive pattern only, although in principle it can be extended through integration with Spark Datalog in the same fashion as edge creation.

[4] version 2.1 at the time of this writing

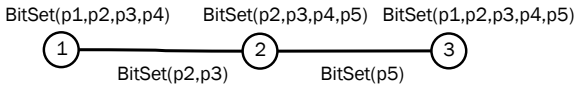


Figure 4.6: OG representation of a small TGraph.

As we will show in Section 5.4, this physical representation is the most efficient for many operations, and especially for those that do not require a join between node and edge RDDs, such as trim and map.

4.4.2 RepresentativeGraph

RepresentativeGraph (RG) is a collection (parallel sequence) of GraphX graphs, one for each representative graph of TGraph, where nodes and edges store the attribute values for the specific time interval, thus using structural locality. This representation supports all operations of TGA that can be expressed over snapshots, i.e., any operation that does not explicitly refer to time. GraphX provides Pregel API to support recursive non-temporal aggregation.

While the RG representation is simple, it is not compact, considering that in many real-world evolving graphs there is a 80% or larger similarity between consecutive snapshots [74]. In a distributed architecture, however, this data structure provides some benefits as operations on it can be easily parallelized by assigning different representative graphs to different workers. Each representative graph in the RG representation is loaded from parquet files using filter pushdown with a time predicate. We include this representation mainly as a naive implementation that serves as a performance baseline.

RG is the most immediate way to implement evolving graphs using GraphX. Without Portal a user wishing to analyze evolving graphs might implement and use the RG approach. However, as we will show in Section 5.4, this would lead to poor performance for most operations.

4.4.3 OneGraph

OneGraph (OG) is the most topologically compact representation and stores all nodes from TV*and* edges from TE once, in a single aggregated data structure. OG emphasizes temporal locality, while also preserving structural locality, but leads to a much denser graph than RG. This, in turn, makes parallelizing computation challenging.

An OG is implemented as a single GraphX graph where the node and edge attributes are bitsets that encode the presence of a node or edge in each time period associated with some representative graph of a TGraph. To construct an OG from on-disk node and edge archives, nodes and edges are

grouped by key and mapped to bits corresponding to periods of change over the graph. Because OG stores information only about graph topology, far fewer periods must be represented and computed for OG than for RG. The actual reduction depends on the rate and nature of graph evolution. Information about time validity is stored together with each node and edge. Figure 4.6 shows a small OG over five intervals.

Similar to ImmortalGraph [74], the analytics are computed over all the representative graphs together. Nodes exchange messages marked with the applicable intervals and a single message may contain several interval values as necessary. This is called a *batching method*, since the computation is batched across time periods.

As we will see experimentally in Section 5.4, OG is often the best-performing data structure for node creation, and also has competitive performance for recursive aggregation (analytics). Because of this focus, OG supports operations only on topology: analytics, node creation, and set operators for graphs with no node or edge attributes. All other operations are supported through inheritance from an abstract parent, and are carried out on the VE data structure. Thus OG and HG, below, can be thought of as indexes on VE.

4.4.4 HybridGraph

In our preliminary experiments we observed that OG exhibited worse than expected performance, especially for large graphs with long lifetimes. The reason this is so is because good graph partitioning becomes difficult as topology changes over time. Communication cost is the main contributor to analytics performance over distributed graphs, so poor partitioning leads to increased communication costs. When the whole graph can fit into memory of a single worker, communication cost goes away and the batching method used by OG becomes the most efficient, as has been previously shown in [74]. To provide better performance on analytics, we introduce **HybridGraph (HG)**. HG trades compactness of OG for better structural locality of RG, by aggregating together several consecutive representative graphs, computing a single OG for each graph group, and storing these as a parallel sequence. In our current implementation each OG in the sequence corresponds to one snapshot group on disk. Thus if no-partitioning snapshot group method is used, HG is equivalent to OG.

Like OG, HG focuses on topology-based analysis, and so does not represent node and edge attributes. HG implements analytics, node creation, and set operators, and supports all other

operations through inheritance from VE. Analytics are implemented similarly to OG, with batching within each graph group.

4.5 Optimizations

4.5.1 Lazy Coalescing

All operations and all sequences of operations must output a valid temporally coalesced TGraph. Several implementations are possible for the coalesce operation over temporal SQL relations, see [13] for details. We use the partitioning method, where the relation (e.g., TV, TE) is grouped by key, and tuples are sorted and folded within each group to produce time periods of maximum length. This involves shuffling between partitions, is computationally expensive, and motivates lazy coalescing.

Correctness of many TGA operations does not depend on whether their input is coalesced. Consequently, Portal supports both eager and lazy coalescing, with lazy being the default for queries that admit it. We now present useful coalescing rules.

As with duplicate elimination, if the coalesced output is expected to be significantly reduced in size, such as after a map operation on a high volatility attribute, performing coalesce eagerly before a time-consuming operation, especially analytics, can be advantageous.

Successive coalescing is unnecessary in TGA just as it is for (non-graph) temporal databases [13]:

$$\mathsf{coal}(\mathsf{coal}(\mathcal{G})) \equiv \mathsf{coal}(\mathcal{G}) \tag{4.1}$$

It is unnecessary to coalesce an already coalesced $\mathcal{G}$, e.g., if the data on disk is known to be coalesced:

$$\mathsf{coal}(\mathcal{G}) \equiv \mathcal{G} \quad \text{iff } \mathsf{is\_coalesced}(\mathcal{G}) \tag{4.2}$$

Trim does not uncoalesce. Trim (Def. 3.2.3) is guaranteed to return a coalesced output when evaluated over a coalesced input. This is because, for any relation R, there will be at most one tuple from R in the result of trim with a validity period that is either the same as it was in R, or further restricted (trimmed). Therefore, there is no need to coalesce $\mathcal{G}$ after trim.

Trim allows lazy coalescing. Trim does not destroy coalescing. Assume x stands for all nontemporal attributes. $\mathsf{trim}_{\boldsymbol{c}}^{T}(\mathcal{G})$ returns tuples $(x|t \cap c)$, for which $T \cap \boldsymbol{c} \neq \emptyset$. If $\mathcal{G}$ is uncoalesced, then there exist some value-equivalent tuples $(x|t_1)$ and $(x|t_2)$ s.t. p_1 meets $t_2 \vee t_1$ contains $t_2 \vee t_1$ overlaps t_2, which would be replaced by $(x, t_1 \cup t_2)$ in coal

$(\mathcal{G})$. Because intersection distributes over union, i.e.,$(t_1 \cup t_2) \cap \boldsymbol{c} = (t_1 \cap \boldsymbol{c}) \cup (t_2 \cap \boldsymbol{c})$, coalescing can be equivalently performed before or after trim:

$$\mathsf{coal}(\mathsf{trim}_{\boldsymbol{c}}^{T}(\mathcal{G})) \equiv \mathsf{trim}_{\boldsymbol{c}}^{T}(\mathsf{coal}(\mathcal{G})) \tag{4.3}$$

Subgraph does not uncoalesce. Vertex-subgraph (Def. 3.2.7) uses a pattern that returns a subset of TV. If TV is coalesced then so is TV′. A similar argument applies to the edges relation TE′ in edge-subgraph.

Temporal subgraph allows lazy coalescing. It was shown in [13] that coalescing can be deferred until after selection if the selection condition is independent of the valid time of the input. When subgraph (Section 3.2.3) is time-invariant, i.e., does not contain temporal predicates, coalescing can be deferred.

$$\mathsf{coal}(\mathsf{subgraph}^{T}(P, \mathcal{G})) \equiv \mathsf{subgraph}^{T}(P, \mathcal{G}) \iff P \text{ has no temporal predicates} \tag{4.4}$$

Vertex- and edge-map may uncoalesce TGraph. Consider computing a vertex-map operation (Def. 3.2.4) $\mathsf{map}_{v}^{T}(m_V : name)\mathcal{G}$ over T1 in Table 3.4. There will be two identical tuples in the result for node v_2 for $[2015/2, 2015/5)$ and $[2015/5, 2015/10)$, which must be coalesced to return a valid TGraph. A similar argument holds for edge-map (Def. 3.2.5).

Vertex- and edge-map allow lazy coalescing, since they processes each input tuple without modifying the corresponding time intervals, but only when they contain no temporal predicates. Map destroys coalescing, and requires to coalesce the output even if the input was eagerly coalesced.

$$\mathsf{coal}(\mathsf{map}_{M_V, M_E}(\mathcal{G})) \equiv \mathsf{coal}(\mathsf{map}_{M_V, M_E}(\mathsf{coal}(\mathcal{G}))) \tag{4.5}$$

Temporal-window node creation requires eager coalescing. Temporal-window node creation allows temporal aggregation by change or time (Def. 3.2.10). When aggregating by change, the input must be coalesced to compute correct aggregation windows. If the input is not coalesced, then there may be two or more consecutive intervals which should be treated as one but would count separately. For aggregation by time, some aggregate functions also require coalesced input. For example, count will produce different result if several consecutive or overlapping value-equivalent

tuples fall within the same window. Thus we add the following conditional coalescing rule:

$$\mathsf{coal}(\mathsf{node}^T_w(w, r_v, r_e, f_{v_1}(k_1), \ldots, f_{v_n}(k_n), f_{e_1}(l_1), \ldots, f_{e_m}(l_m), \mathcal{G})) \equiv \mathsf{coal}(\mathsf{node}^T_w(w, r_v, r_e, f_{v_1}(k_1), \ldots, f_{v_n}(k_n), f_{e_1}(l_1), \ldots, f_{e_m}(l_m), \mathcal{G})) \quad (4.6)$$

Temporal intersection, union, and difference allow lazy coalescing. Temporal graph intersection (Def 3.2.12) is a join followed by the resolve primitive that includes temporal aggregation. As shown in [13], coalescing can be deferred until after a join. For temporal aggregation, which is not addressed in [13], the same argument can be made as for TRA union and intersection, namely, that it is time-invariant. Thus we can defer coalescing until after temporal graph intersection. Temporal graph union (Def. 3.2.11) uses an outer join, and the same argument applies. Temporal difference (Def. 3.2.13) uses a left outer join, and the same argument applies. Note that intersection and union operations may destroy coalescing due to the use of aggregation, and so a final coalesce is required over the result even if inputs were eagerly coalesced.

$$\mathsf{coal}(\mathsf{coal}(\mathcal{G}_1) \cap^{TG} \mathsf{coal}(\mathcal{G}_2)) \equiv \mathsf{coal}(\mathcal{G}_1 \cap^{TG} \mathcal{G}_2) \quad (4.7)$$

$$\mathsf{coal}(\mathsf{coal}(\mathcal{G}_1) \cup^{TG} \mathsf{coal}(\mathcal{G}_2)) \equiv \mathsf{coal}(\mathcal{G}_1 \cup^{TG} \mathcal{G}_2) \quad (4.8)$$

$$\mathsf{coal}(\mathsf{coal}(\mathcal{G}_1) \setminus^{TG} \mathsf{coal}(\mathcal{G}_2)) \equiv \mathsf{coal}(\mathcal{G}_1 \setminus^{TG} \mathcal{G}_2) \quad (4.9)$$

Temporal user-defined analytics allow lazy coalescing. Analytices are a special case of temporal graph aggregation (Section 3.2.4). Analytics are time-invariant functions applied to each representative graph. Thus when applied to value-equivalent graphs, they will produce equivalent results, and so coalesce can be deferred. However, analytics may destroy coalescing because the same aggregate may be produced for a node over successive periods, and so it is required to coalesce over the final result. If uda is an analytic, then:

$$\mathsf{uda}(\mathcal{G}) \equiv \mathsf{uda}(\mathsf{coal}(\mathcal{G})) \quad (4.10)$$

4.5.2 Lazy Referential Integrity Enforcement

All TGA operations must enforce the foreign key constraint from TE to TV. However, foreign key constraint enforcement is an expensive operation that requires a join. Many TGA operations

do not modify TE, or modify it in such a way that application of the constrain primitive is not required.

Trim does not require FK enforcement. To see why, consider an edge $\mathsf{TE}(e, v_1, v_2, a|t_1)$ and one of the corresponding nodes $\mathsf{TV}(v_1, a|t_2)$, such that t_2 contains t_1 (per Definition 3.1.3 requirement R2). Suppose now that trim was applied to TV and to TE with interval $\boldsymbol{c}$. Is it possible that edge $(e, v_1, v_2, a|t_1 \cap \boldsymbol{c})$ is in the result of $\mathsf{trim}_{\boldsymbol{c}}^{T}(\mathcal{G}).\mathsf{TE}$ (i.e., $t_1 \cap \boldsymbol{c} \neq \emptyset$), while node $\mathsf{TV}(v_1, a, t_2 \cap \boldsymbol{c})$ is not in the result of $\mathsf{trim}_{\boldsymbol{c}}^{T}(\mathcal{G})$ (i.e., $p_2 \cap c = \emptyset$)? Clearly, the answer is no, since t_2 contains t_1, and so it must be the case that t_2 contains $(t_1 \cap \boldsymbol{c})$.

Vertex- and edge-map do not require FK enforcement. This is because the timestamps are not modified by the operation.

Vertex-subgraph requires FK enforcement. A pattern specifies a selection condition over the nodes, and computes the vertex-induced subgraph. In this case we cannot compute TE' from TE alone, but will need to remove edges for which one or both nodes are not present in TV'.

Edge-subgraph does not require FK enforcement. This is because edge-subgraph does not modify TV and takes a subset of TE.

Aggregation does not require FK enforcement. This is because only attributes of TV are modified, but not the node periods of validity.

Temporal-window node creation requires FK enforcement in the general case. As illustrated in Example 34, an edge may be reduced in duration in the output because a node is removed based on the quantifier r_v.

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

FK enforcement is only required if r_v is more restrictive than r_e. To see why, consider an interval w, one of the windows computed based on specification W. Edge quantifier r_e is translated into a ratio r. We produce an edge $(e, v_1, v_2|w)$ iff $\exists\{(e, v_1, v_2|p_1), \ldots, (e, v_1, v_2|p_k)\} \in \mathsf{TE} \mid (\bigcup_{i=1}^{k} t_i \cap w)/w > r$. In other words, an edge is only produced if the ratio of the sum of the intervals of e in window w to the length of the window w is greater than the ratio r. According to Definition 3.1.3, requirement R2, if the input TGraph is valid, then we must also produce $(v_1|w)$ and $(v_2|w)$ if $r_e \leq r_e$,

since node periods are supersets of edge periods. However, if $r_e > r_e$, we may not produce a node that meets r_e but does not meet the less restrictive r_v.

Union does not require FK enforcement. Temporal full outer join produces a node tuple for all periods in TV_1 and TV_2. If $\mathcal{G}_1$ and $\mathcal{G}_2$ are valid graphs, then referential integrity holds for TE'.

Intersection does not require FK enforcement. Consider an edge $\mathsf{TE}(e, v_1, v_2, a|t^e) \in \mathsf{TE}'$ and one of the corresponding nodes $\mathsf{TV}(v_1, a|t^v) \in \mathsf{TV}'$ computed by intersection. To violate the FK constraint on TE', there must exist a time instant $t \in t^e \wedge t \notin t^v$. But that is not possible, since by definition of temporal join, e exists in both TE_1 and TE_2, and $\mathcal{G}_1$ and $\mathcal{G}_2$ are valid TGraphs.

Difference requires FK enforcement. Consider an edge $\mathsf{TE}(e, v_1, v_2, a|t)$ that exists in $\mathcal{G}_1$ but not in $\mathcal{G}_2$ and node v_1 that exists in both $\mathcal{G}_1$ and $\mathcal{G}_2$. According to Definition 3.2.13, node v_1 will not be produced in TV'. This invalidates edge e, which must be removed.

Edge creation requires FK enforcement. Edge creation (Def. 3.2.14) generates new nodes in TE' that are not present in TE. It must be verified that these new edges are valid in relation to the periods of validify of its two end points.

4.5.3 Trim Pushdown

As described in Section 4.3, the on-disk data format supports filter pushdown by time. For ease of use, we provide a GraphLoader utility that can initialize any of the four physical representations from Apache Parquet files on HDFS or on local disk. This utility accepts a date range and filters the data first by snapshot group to only include snapshot groups in the desired range, and then through Parquet filter pushdown. For datasets with long evolution history this optimization provides a substantial performance improvement.

4.6 Limitations

The main limitation of the current prototype is the lack of support for full functinoality with respect to temporal navigational graph patterns in aggregation and attribute-based node creation. This can be supported through the same mechanism as extended vertex- and edge subgraph operations, i.e., through integration with Spark Datalog [82].

Another limitation, already discussed above, is with Skolem functions and node/edge ids. All ids are of the long data type. Currently the prototype pushes the responsibility for id creation to a user-provided Skolem function. However, the distributed nature of the system makes implementing

such a function correctly challenging. One approach a user may take is to take the attribute by which the nodes are being created, e.g., school, and assign the ids based on its hash code. As with any hashing function, this does not guarantee correctness because of the chance of collision. Another possible approach is to use the accumulators mechanism in Spark, but we have not tested the effectiveness of such an approach.

Finally, Portal currently does not provide query optimization. The responsibility for picking the best in-memory representation and the best sequence of operations falls to the user. We are in the early stages of implementing query optimization now.

Chapter 5: Evaluation

In the course of Portal development we conducted multiple experiments to evaluate the effectiveness of the physical layouts (Section 5.2) and different in-memory representations (Section 5.4). We also evaluate Portal in comparison to the published baselines, G* and ImmortalGraph, and report the results in Section 5.5. Finally, we close the loop on our motivating examples in Section 1.3 by implementing them using Portal API and report the results in Section 5.6.

The results of our experiments show that Portal scales to large datasets even on a modest size cluster and outperforms the G* baseline, while providing additional functionality not available in the baseline.

5.1 Setup

All experiments were conducted on a 16-slave in-house Open Stack cloud, using Linux Ubuntu 14.04 and Spark v2.0. Each node has 4 cores and 16 GB of RAM. Spark Standalone cluster manager and Hadoop 2.6 were used.

Because Spark is a lazy evaluation system, a materialize operation was appended to the end of each query, which consisted of the count of nodes and edges. Each experiment was conducted 3 times with a cold start – the running time includes the setup time of submitting the job to the cluster manager, uploading the jar to the cluster, reading the data from disk, building the chosen representation, and running a single query. 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. No computation results were shared between subsequent runs.

Table 5.1: Experimental datasets.

Dataset	**|V|**	**|E|**	**Time Span**	**Evol. Rate**
wiki-talk-en	2.9M	10.7M	2002–2016	14.4
nGrams	88M	2.85B	1520–2009	16.67
Twitter	505.4M	23B	2006–2012	88

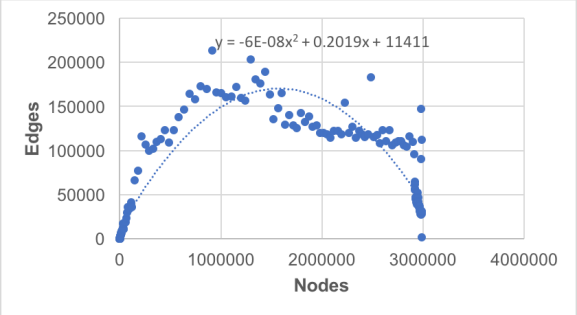


Figure 5.1: The wiki-talk dataset exhibits a quadratic relationship between the number of nodes and edges.

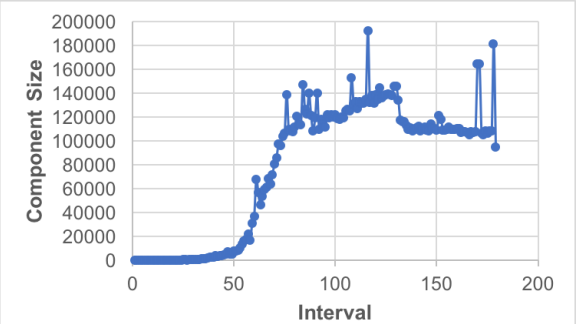


Figure 5.2: The size of the largest connected component in the wiki-talk dataset.

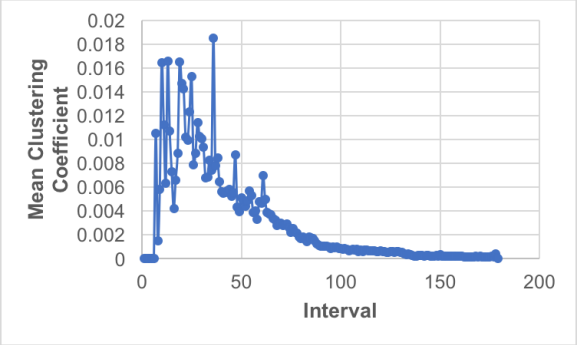


Figure 5.3: The graph clustering coefficient of the wiki-talk dataset over time.

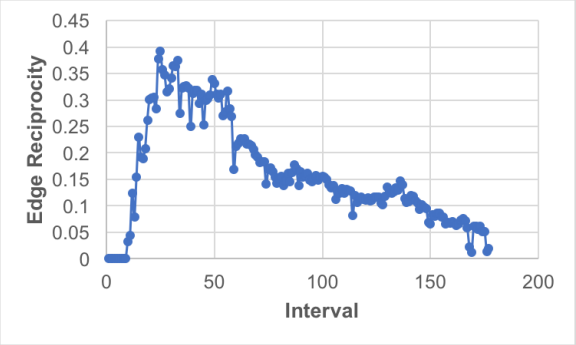


Figure 5.4: The edge reciprocity of the wiki-talk dataset over time.

Data. We evaluate performance of our framework on three real open-source datasets, summarized in Table 5.1. The datasets differ in size, in the number and type of attributes and in evolution rates, calculated as the average graph edit similarity [77]. Edit similarity between any two snapshots is calculated as the ratio of the common edges between two graphs to the sum of edges:

$2 * |E_1 \cap E_2|/(|E_1| + |E_2|)$

wiki-talk (`http://dx.doi.org/10.5281/zenodo.49561`) contains over 10 million messaging events among 3 million wiki-en users, aggregated at 1-month resolution. Wiki-talk is a very sparse dataset with short-lived edges, so it does not exhibit the expected exponential growth of the number of edges relative to the number of nodes (Figure 5.1). The size of the largest weakly connected component grows over time but stabilizes at roughly 110K nodes, about 4% of the total size, as can be seen in Figure 5.2. Due to extreme graph sparcity, the graph mean clustering coefficient is extremely small and diminishes over time, approaching 0 (Figure 5.3). Similarly, the edge reciprocity is also small and diminishes over time, from the maximum of about 40% to about 5% toward the end of its lifetime (Figure 5.4).

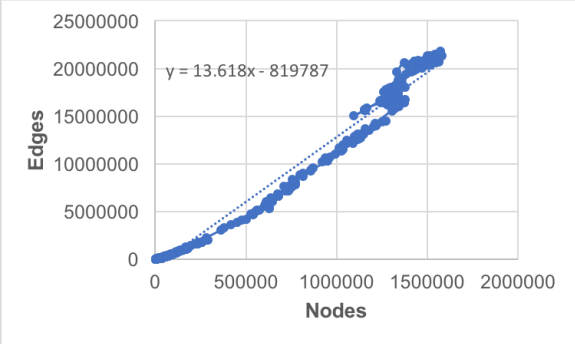


Figure 5.5: The nGrams dataset exhibits a linear relationship between the number of nodes and edges.

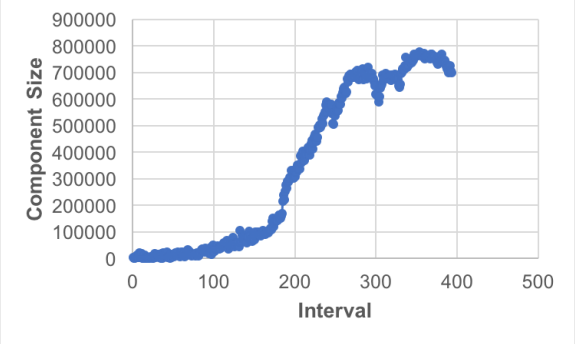


Figure 5.6: The size of the largest connected component in the nGrams dataset.

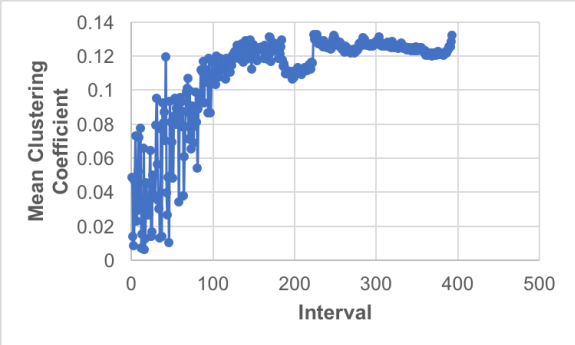


Figure 5.7: The graph clustering coefficient of the nGrams dataset over time.

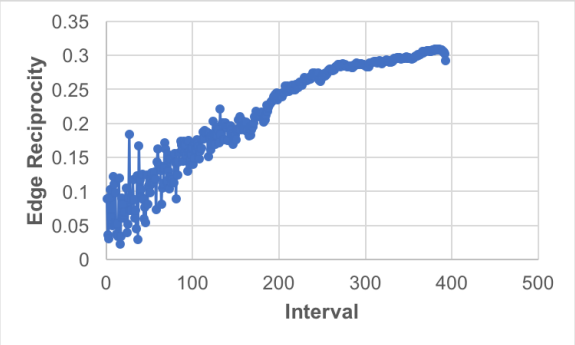


Figure 5.8: The edge reciprocity of the nGrams dataset over time.

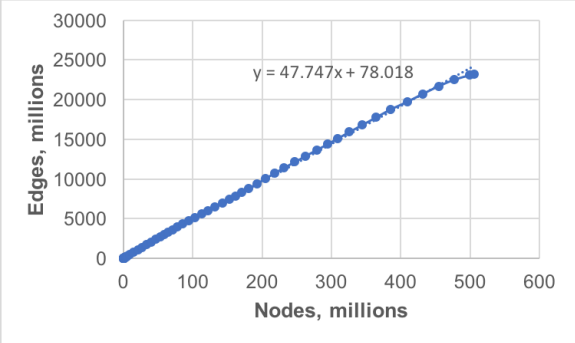


Figure 5.9: The Twitter dataset exhibits a linear relationship between the number of nodes and edges.

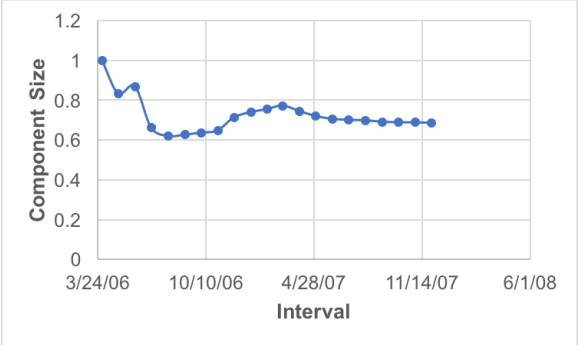


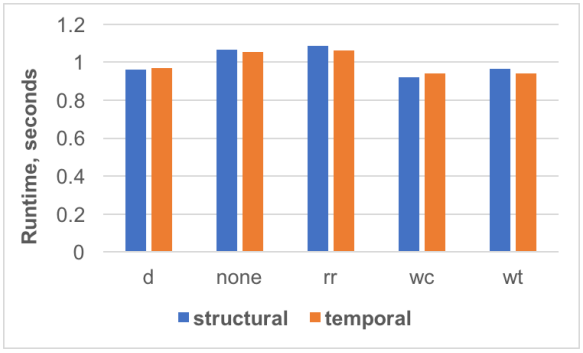
Figure 5.10: The size of the largest connected component in the Twitter dataset.

nGrams (`http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`) contains word co-occurrence pairs, with 88 million word nodes and over 2.8 billion undirected edges. While not as sparse a graph as the wiki-talk, the edges still can appear and go away. The relationship between the node and edge counts is linear, not exponential, as can be seen in Figure 5.5. The size of the largest weakly connected component increases with time to about 50% of all nodes (Figure 5.6), so it is substantially more connected than the wiki-talk dataset. It also spans more than twice as many time intervals. The mean clustering coefficient is initially chaotic, but over time stabilizes around 0.12, as can be seen in Figure 5.7. The edge reciprocity also increases over time as the graph densifies, and stabilizes at about 0.3 (Figure 5.8).

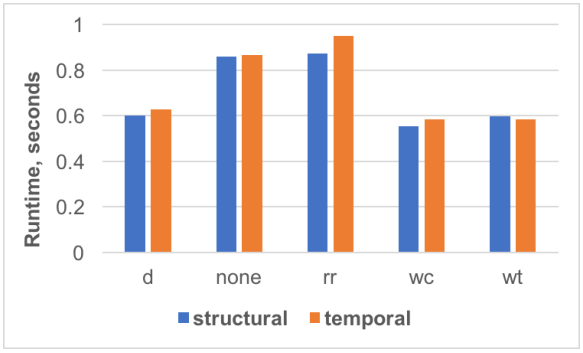
The Twitter social graph [39] contains over 23 billion directed follower relationships between 0.5 billion Twitter users collected in 2012, sampled at 1-month resolution based on account creation information from April 2006. Surprisingly, even though this dataset is growth-only, i.e., every node and edge is added once and never goes away, the densification still exhibits a linear rather than an exponential trend, as shown in Figure 5.9. Nevertheless, the Twitter graph is very dense and connected from the very beginning, and the largest weakly connected component contains about 3/4 of all nodes (Figure 5.10).

5.2 Physical layout

We evaluate the effectiveness of the structural and temporal locality and different snapshot group partition methods (refer to Section 4.3) on local and global queries. The temporal locality outperforms structural locality in all experiments due to the lack of sufficient discrimination in the time range in the test datasets. The equi-depth partition methods of snapshot groups is the most effective for local and global point queries, but no-partition method outperforms all others on the

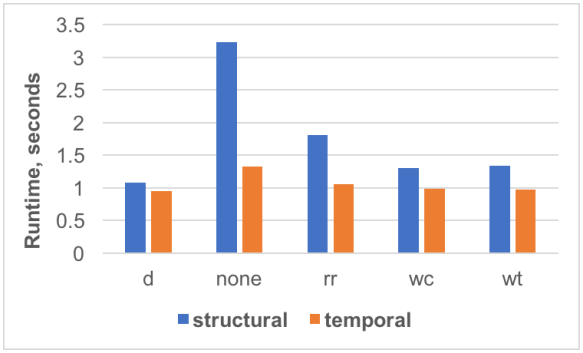


(a) nodes

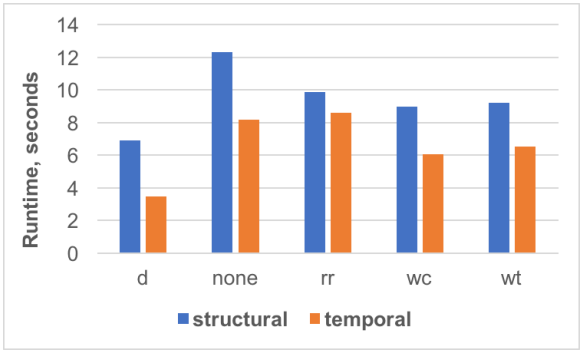


(b) edges

Figure 5.11: Performance of each snapshot group and locality format on local point queries on the wiki-talk dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method.



(a) nodes



(b) edges

Figure 5.12: Performance of each snapshot group and locality format on local point queries on the nGrams dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method.

local range queries. Based on the results of these experiments we selected the no-partition and the equi-depth partition methods with the temporal locality for all subsequent experiments.

5.2.1 Local Point Queries

Local point queries are queries to retrieve a single node or an edge, and its attribute, at a specific time instant. We used skewed sampling to retrieve a list of 100 nodes and 100 edges spread across the dataset evolution period. By skewed we mean that the time instant distribution of the query set is not uniform but rather an approximation of the dataset distribution, where more tuples exist at later dates.

We compare two types of locality layout: structural and temporal. The structural locality layout places together nodes/edges that co-exist in time and is achieved with a primary sort by the start time of the period of validity. The temporal locality layout places together the full history of change for each node/edge and is achieved with a primary sort by the node/edge id.

We evaluate all five snapshot group partition methods: no-partitioning, i.e., a single group, two variants of the equi-width partitioning, the equi-depth partitioning, which aims to balance the sizes of the snapshot groups, and the redundancy partitioning, which aims to limit the redundancy between snapshot groups.

We report an average single point query. For wiki-talk, the difference between different layouts and snapshot group partition methods is negligible for both nodes and edges, as can be seen in Figure 5.11. This can be explained by the fact that wiki-talk is a small dataset. In fact, any differences between partition methods result from the total number of snapshot groups: for the two slowest methods, none and redundancy (rr), there are 1 and 5 snapshot groups, respectively, compared to 15 groups for equi-width (wt), equi-width by change (wc), and equi-depth (d). The difference between the two types of locality is also negligible. Wiki-talk has a single tuple per node, with no change over the graph history, while each edge is short-lived.

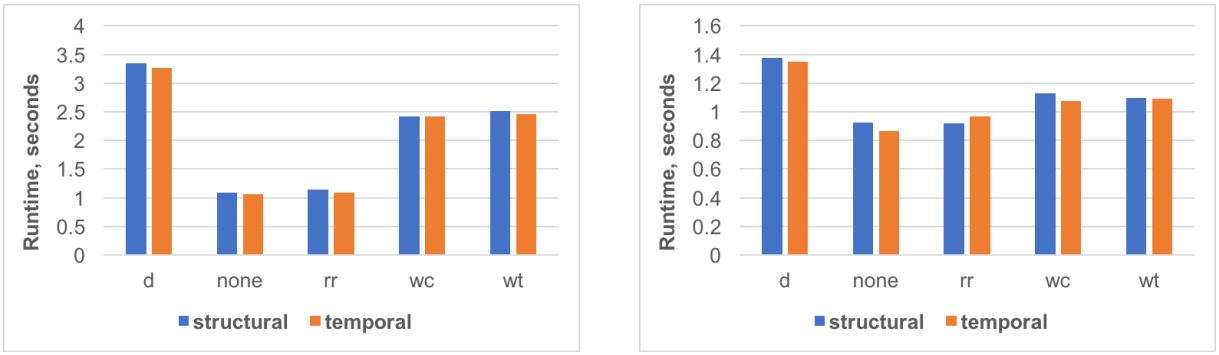
Results from the nGrams dataset are more useful, as can be seen in Figure 5.12. The average single query time is slower than in the wiki-talk dataset, for both nodes and edges, as can be expected due to the increased dataset size (nGrams has 30x number of nodes than wiki-talk, and 250x number of edges). However, the best performing partition method, equi-depth, with temporal locality, is only a fraction of a second slower for nodes and 6 times slower for edges. Additionally, it can be seen that the temporal locality outperforms the structural locality in all cases, in some cases by a factor of two. Recall that the temporal locality sorts by the id first and places all the tuples for a single node or edge together. Thus filter pushdown on the id is more effective than when the data is sorted by date first.

The nGrams dataset results indicate that the temporal locality is more effective than the structural locality, and the equi-depth partition method is as, or more, effective as all other methods, especially as the data size increases.

5.2.2 Local Range Queries

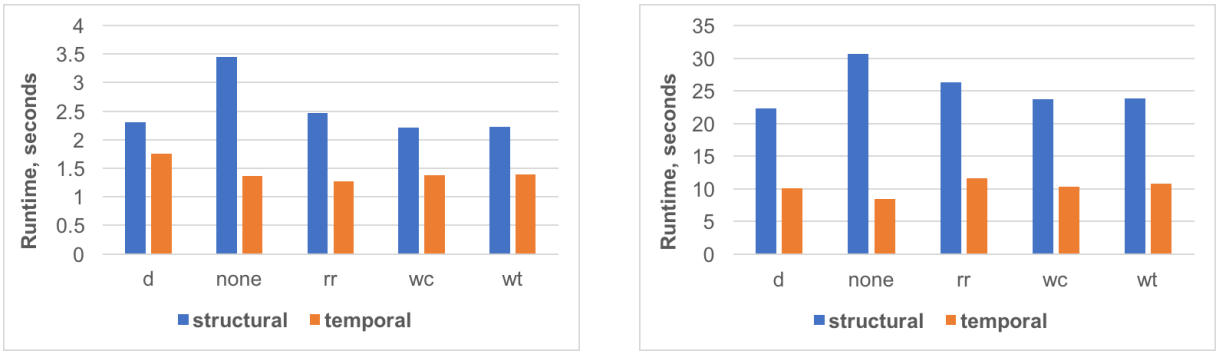
Local range queries are queries to retrieve a history of a node or an edge over the interval of interest. Similarly to local point queries, local range queries were executed using skewed sampling, averaging over 100 node (resp. edge) queries. Figure 5.13 shows the result for node and edge queries over the wiki-talk dataset, and Figure 5.14 over the nGrams dataset.

As with the local point queries, there is no difference between the performance of the two types of locality on the wiki-talk dataset. However, the no-partitioning method is a clear winner for nodes



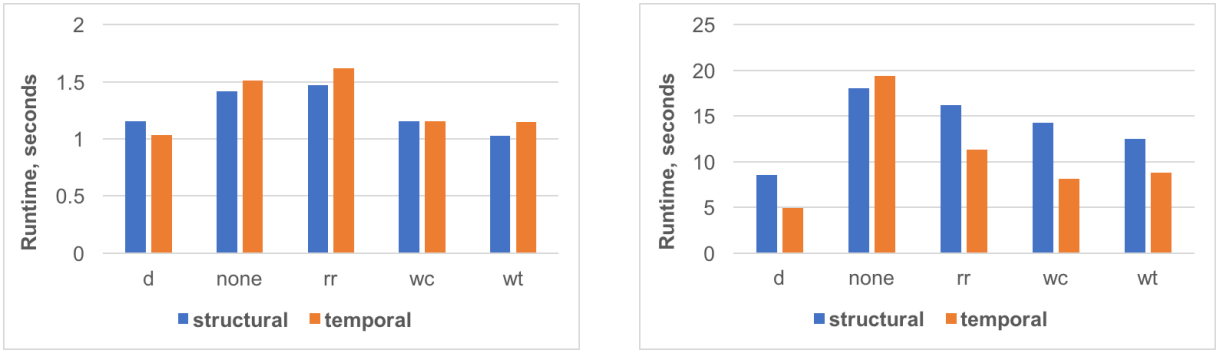
(a) nodes (b) edges

Figure 5.13: Performance of each snapshot group and locality format on local range queries on the wiki-talk dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method.



(a) nodes (b) edges

Figure 5.14: Performance of each snapshot group and locality format on local range queries on the nGrams dataset. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method.



(a) on wiki-talk

(b) on nGrams

Figure 5.15: Performance of each snapshot group and locality format on global point queries. d = equi-depth partition method, none = no-partitioning, rr = redundancy partition method, wc = equi-width by change partition method, wt = equi-width partition method.

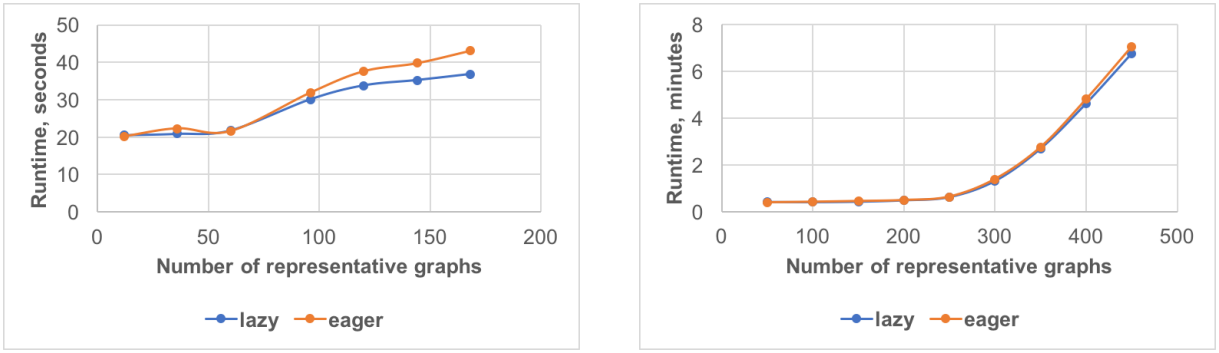
and edges. This again can be explained by the nature of the wiki-talk evolution: there is no change for wiki-talk nodes, which means that a single node tuple is split into as many tuples as there are groups or nearly so, depending on when it appeared. As a result, the local range query has to retrieve only a single tuple using the no-partitioning method, whereas with all other methods it retrieves several, depending on the duration of the query interval.

The nGrams dataset results indicate that the temporal locality is significantly more effective for local range queries than the structural locality, and the no-partitioning method is more effective than all other methods.

5.2.3 Global Point Queries

Global point queries are queries to retrieve a single snapshot at a specific time point. We sampled each dataset to get a set of 100 dates approximating the distribution of all the dates in the dataset. We report an average single global point query.

In wiki-talk (Figure 5.15a) there is no significant difference between the two types of locality, and the difference between different partition methods is negligible as well. However, similarly to the local point and range experiments, temporal locality outperforms structural locality on the nGrams dataset (Figure 5.15b). This result is unexpected, considering that there is no filter pushdown by id in the global point query execution. We conclude that the spread of the period start time is so small as to render sorting first by the time attribute undesirable. The wiki-talk dataset has only 179 distinct start times across all nodes and edges, and nGrams only about 400. While a different dataset may have a wider spread of start times, any large graph is expected to have orders



(a) on wiki-talk

(b) on nGrams

Figure 5.16: Performance of the trim operation with eager and lazy coalescing.

of magnitude more nodes than distinct start times. This explains why the structural locality, as designed, cannot outperform the temporal locality.

As with the local point queries, the depth partitioning method yields the best performance for global point queries on nGrams.

5.3 Optimizations

We introduced three optimizations in the Portal prototype, described in Section 4.5: lazy coalescing (Section 4.5.1), lazy referential integrity enforcement (Section 4.5.2), and filter pushdown (Section 4.5.3). In this section we show the effect of lazy coalescing and filter pushdown on the system performance. Lazy referential integrity enforcement is in effect at all times.

Lazy coalescing is always beneficial in our experiments, regardless of the rate of coalescing. In the extreme case the coalesced TGraph has significantly fewer nodes or edges. Even in this case the topology of the TGraph does not change, and so analytics do not benefit from eager coalescing.

Filter pushdown is significantly beneficial when the trim period is small compared to the overall TGraph history period.

5.3.1 Lazy Coalescing

We evaluate the eager vs. lazy coalescing by taking a noncoalesced input from equi-depth snapshot groups and executing a trim operation over it. With eager coalescing the input is coalesced immediately upon load, and then the trim operation is carried out. Since trim is known to not uncoalesce, no additional coalescing is carried out. With lazy coalescing the input is coalesced only at the end, after trim is carried out. With this setup we expect no difference in performance

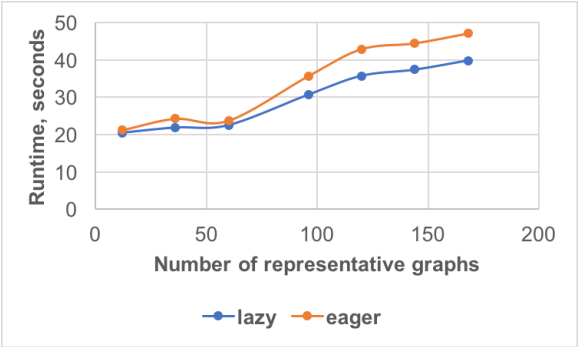
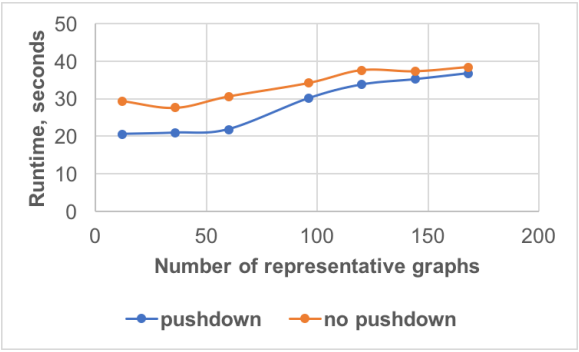
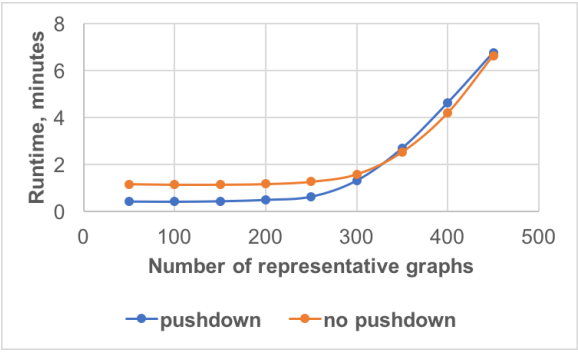


Figure 5.17: Performance of the map operation with eager and lazy coalescing.



(a) on wiki-talk



(b) on nGrams

Figure 5.18: Performance of the trim operation with filter pushdown.

between lazy and eager coalescing when filter pushdown is in effect, which is what the data shows (Figure 5.16). Any differences between the two conditions are not statistically significant.

We also evaluate the eager vs. lazy coalescing by taking a noncoalesced input and executing a vertex-map operation over it, which is known to uncoalesce. In this case eager coalescing condition performs the coalesce operation twice, once on load and once at the end, whereas the lazy coalescing performs it only at the end. The lazy coalescing condition carries out fewer operations and is expected to outperform the eager coalescing, which is what the data shows (Figure 5.17). The experiment also shows that the cost of coalescing is linear to data size, which makes sense, considering that coalescing is supported by a group-by operation.

5.3.2 Filter Pushdown

Recollect that on-disk data is stored in snapshot groups representing different temporal segments of the overall evolution, and is furthermore sorted by id and time to take advantage of the SparkSQL filter pushdown functionality.

We evaluate the effectiveness of the filter pushdown by executing the trim operation with and without the pushdown over varying time periods. As expected, filter pushdown is beneficial for

small periods relative to the overall timeline (Figure 5.18). As the trim period approaches the overall size, it takes the same amount of time with and without pushdown, on average, since most of the nodes and edges match.

5.4 In-Memory Representations

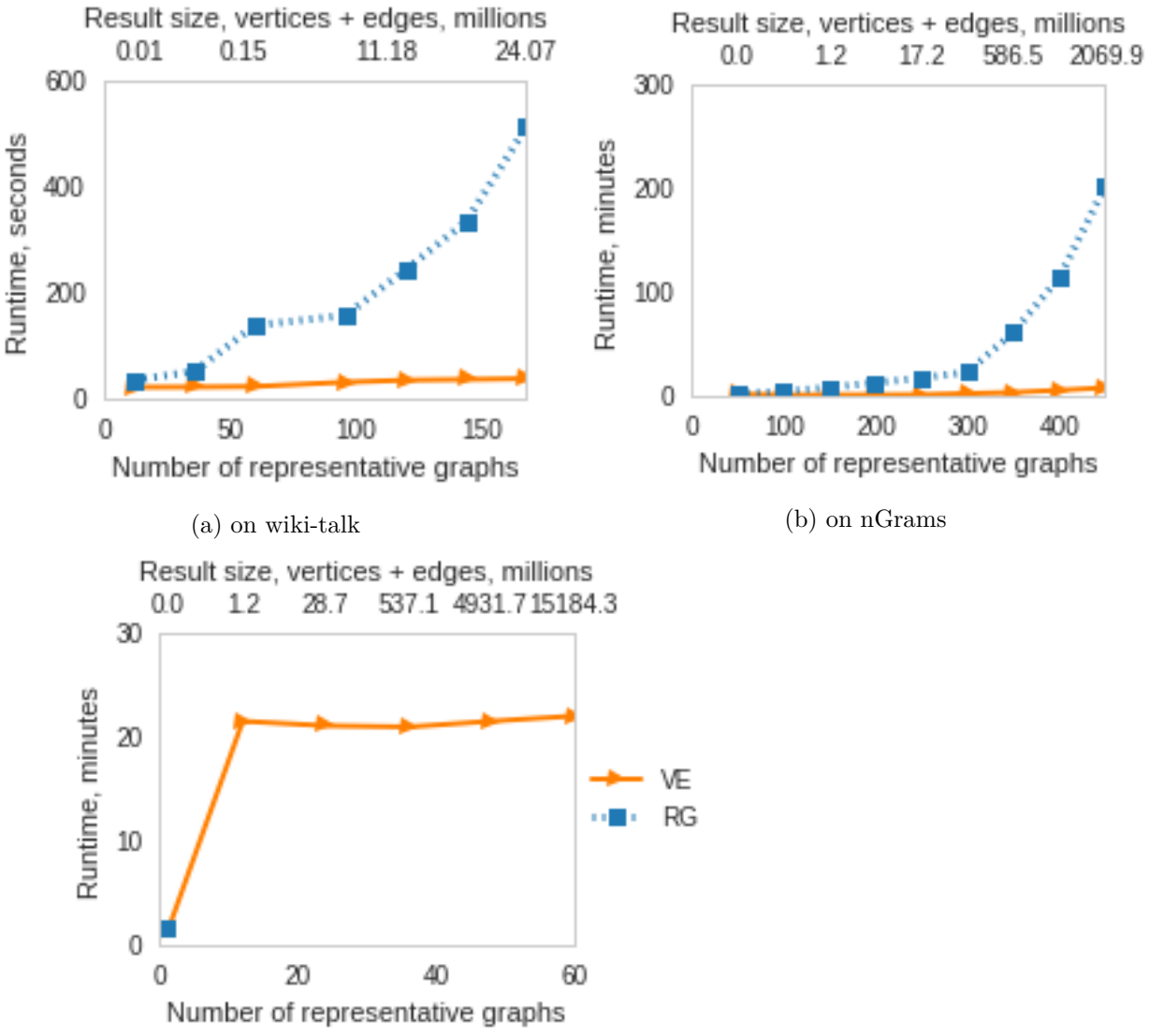
In this section we evaluate the performance of different in-memory representations on TGA operators, including trim (Section 5.4.1), vertex-map (Section 5.4.2), vertex-subgraph (Section 5.4.3), nonrecursive and recursive aggregation (Section 5.4.4), node creation (Section 5.4.5), and the binary set operators union, intersection, and difference (Section 5.4.6). We do not include the results of the evaluation of edge-map and edge-subgraph as uninteresting, since they are no different than the vertex variants. The evaluation of the edge creation operator is pending.

No one single representation provides the best performance for all operations. The RG representation, included primarily as a baseline, does not perform well on all but the smallest cases and should not be considered in the future. The VE and HG representations provide good performance in most cases, although performance degrades when a very large join is required for foreign key enforcement.

5.4.1 Trim

trim (Def. 3.2.3) performance was evaluated by varying the input time window and materializing the TGraph, and is presented in Figures 5.19a for wiki-talk, 5.19b for nGrams, and 5.19c for Twitter. trim is expected to be more efficient when executed over VE (Section 4.4.1) when data is coalesced on disk than over RG (Section 4.4.2), and we observe this in our experiments. This is because multiple passes over the data are required for RG to compute each representative graph, leading to linear growth in running times, even with filter pushdown. trim over VE simply executes temporal selection with filter pushdown and has linearly increasing running times as the output size increaes. This experiment essentially measures the cost of materializing RG from its on-disk representation. We observed the same linear trend in our preliminary work, when the data was stored as individual snapshots on disk, although less redundant work is needed in that case.

The results of trim on the Twitter dataset demonstrate the limits of effectiveness of the temporal locality and the equi-depth partitioning. The performance of VE is on the order of minutes, and RG is unable to compute within two hours for all but the smallest date range. The explanation lies in the nature of the Twitter dataset. The Twitter dataset is a very large growth-only dataset.



(a) on wiki-talk

(b) on nGrams

(c) on Twitter

Figure 5.19: Performance of trim operator over varying intervals.

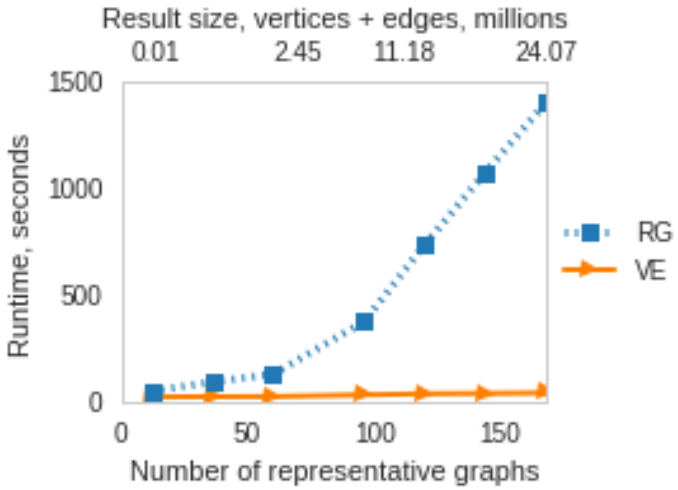


Figure 5.20: Performance of vertex-map operator over varying intervals on wiki-talk.

Nodes and edges in this dataset are added over time, but are never removed or updated. At the same time, with temporal resolution of 1 month, there are only 72 distinct start times. The equi-depth partition method cannot divide a growth-only dataset into more than one snapshot group as the minimum partition load is always the total number of tuples. The trim operation relies both on the filter pushdown and the ability to select a small subset of snapshot groups. Neither works effectively in this case.

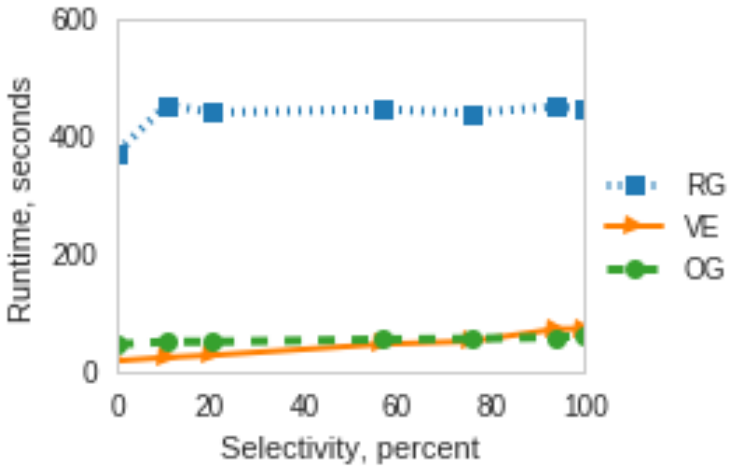
The VE representation provides good performance on trim in most cases, but relies on the effectiveness of the filter pushdown mechanism.

5.4.2 Map

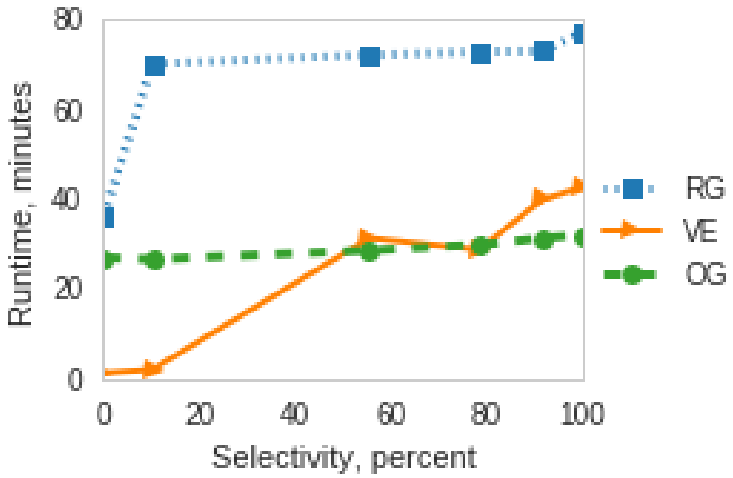
We evaluate vertex-map (Def. 3.2.4) performance by varying the data size through the trim operation. vertex-map exhibits a similar trend as trim: linear running time for VE and a linear increase in the running time with an increasing number of representative graphs for RG (Figure 5.20 for wiki-talk, similar for other datasets). Performance of map is not distinguishable from that of trim when run with the equi-depth partition method because both map and trim must coalesce its output as the last step. We do not include the results of the edge-map operation (Def 3.2.5) here as uninteresting, since it is expected to be no different than the vertex-map based on its definition.

5.4.3 Vertex Subgraph

vertex-subgraph (Def. 3.2.7) performance was evaluated by specifying a condition on the $length(prop) < t$ of the node property prop, with different values of t leading to different se-



(a) on wiki-talk



(b) on nGrams

Figure 5.21: Performance of vertex-subgraph operator over varying selectivity.

lectivity. This experiment was executed for wiki-talk (with *username* as the property) and for nGrams (with *word* as the property). Twitter has no vertex attributes and was not used in this experiment.

Figure 5.21a shows performance for RG and VE on the wiki-talk dataset, and Figure 5.21b on the nGrams dataset. Performance of RG is a function of the number of intervals and is insensitive to the selectivity. The behavior on VE is dominated by FK enforcement: with high selectivity (few nodes) broadcast join affords performance linear in the number of edges, whereas for a large number of nodes broadcast join is infeasible and a hash-join is used instead, which is substantially slower. VE provides an order of magnitude better performance than RG on wiki-talk: up to 1.5 min for VE, in contrast to between 5 and 12 minutes for RG. On nGrams, broadcast join is infeasible for any selectivity except 0, and VE performance is about 2x better than RG performance, 40 min on average for VE vs. about 80 min for RG.

In addition, we evaluated the performance of OG on this operation, a variant where all node attributes are stored with the node in a single GraphX graph. OG achieves somewhat better performance than VE, but not substantially so, as it also internally must perform a join in order to remove invalid edges. Since nGrams has upwards of 2 billion edges, a join is expected to be relatively slow.

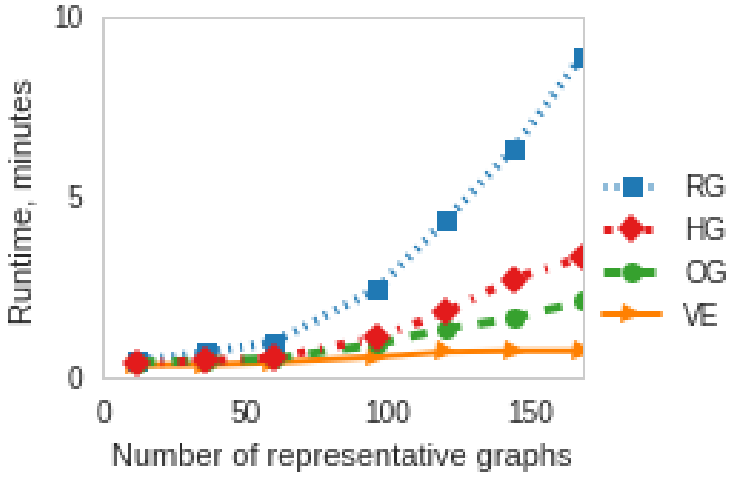
In summary, OG provides the best performance on vertex-subgraph due to the cost of foreign key enforcement.

5.4.4 Aggregation

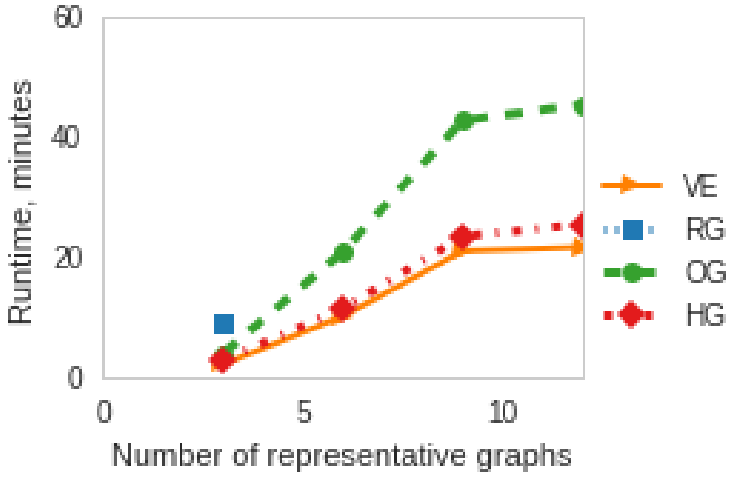
aggregation (Def. 3.2.8) performance was evaluated on all representations with a computation of vertex degrees, varying the size of the temporal window obtained with trim. The results in Figure 5.22b indicate that materialization of each representative graph required for RG makes it not a viable candidate for this operation, especially over large datasets. VE, OG, and HG exhibit linear increase in performance as the trim size is increased, with VE the most efficient. A comparison with the runtime of trim only (Figure 5.19a) indicates that the additional cost of computing degrees is negligible compared to the cost of loading the data from HDFS (21.54 min for 12 RGs with VE with trim only vs. 21.62 min with VE with trim and degrees).

Similar performance is observed in the wiki-talk dataset (Figure 5.22a).

Recall that whole-graph computation over each snapshot, i.e., snapshot analytics, are a special case of the aggregation operator (Def. 3.2.8). We implemented PageRank (PR) and Connected

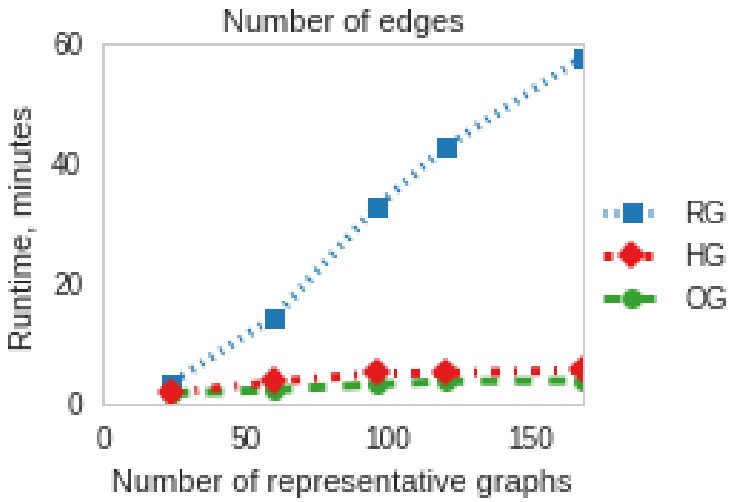


(a) on wiki-talk

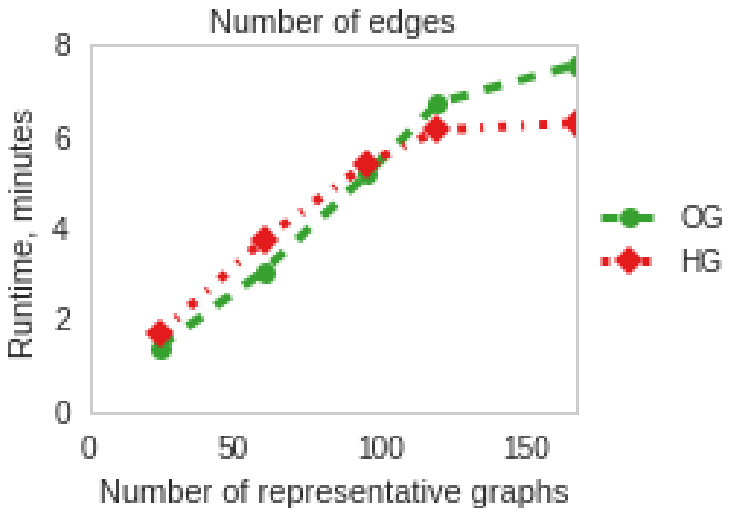


(b) on Twitter

Figure 5.22: Performance of aggregation operator over varying intervals, computing node degrees. $\mathsf{agg}_p^T(\mathsf{trim}_t^T(\mathcal{G}))$

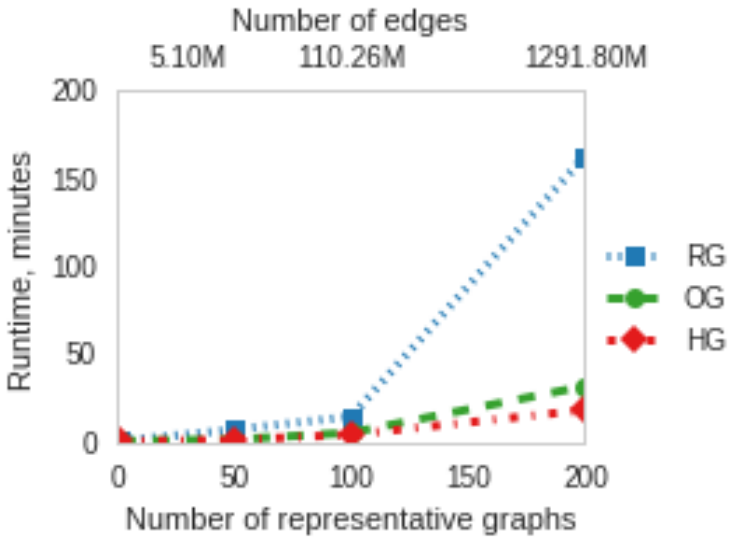


(a) connected components

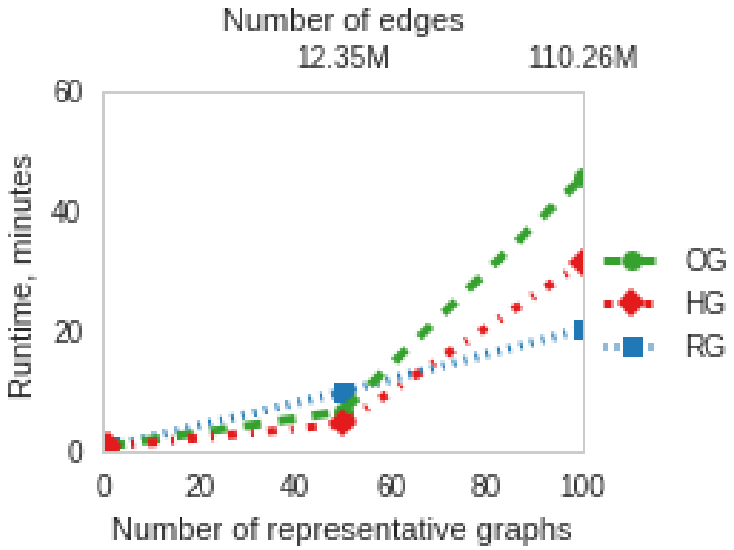


(b) PageRank

Figure 5.23: Performance of snapshot analytics over varying time intervals on the wiki-talk dataset.

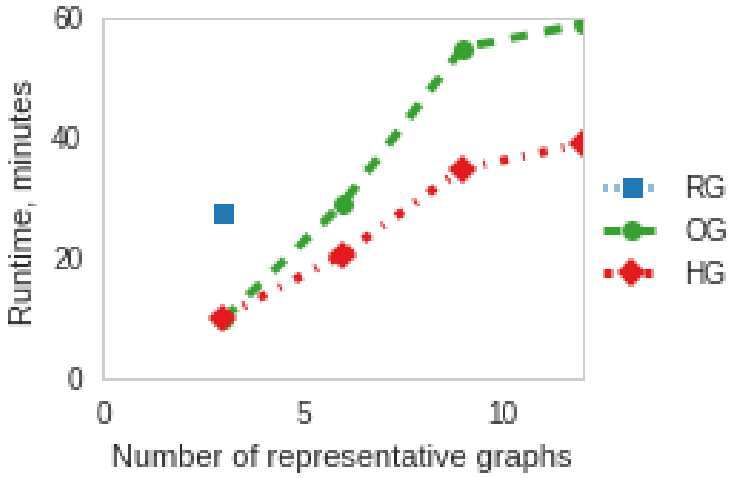


(a) connected components

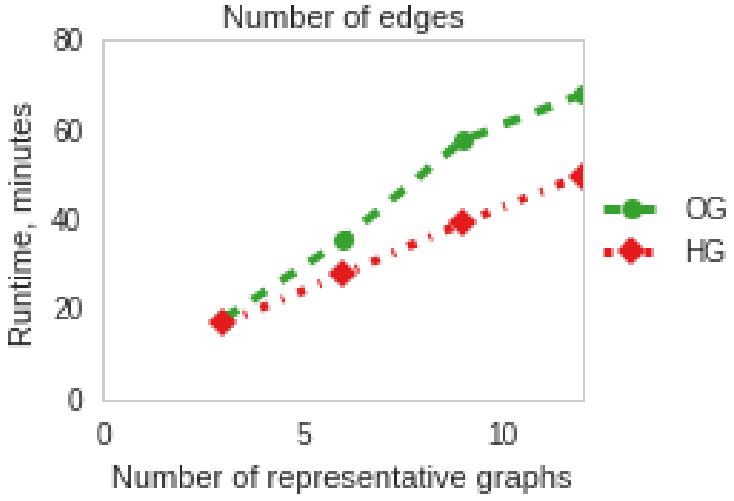


(b) PageRank

Figure 5.24: Performance of snapshot analytics over varying time intervals on the nGrams dataset.



(a) connected components



(b) PageRank

Figure 5.25: Performance of snapshot analytics over varying time intervals on the Twitter dataset.

Components (CC) analytics for the three graph-based representations using the Pregel GraphX API (see discussion in Section 4.4). PR was executed for 10 iterations or until convergence, whichever came first. CC was executed until convergence with no limit on the number of iterations. Performance of Pregel-based algorithms depends heavily on the partitioning strategy, with best results achieved where cross-partition communication is small [96]. For this reason, we evaluated only with the E2D strategy.

Performance was evaluated on time slices of varying size. Since analytics are essentially multiple rounds of aggregate operations, the performance we observe is an amplified version of aggregate performance. For a very small number of graphs (1-2), RG provides good performance, but slows down linearly as the number of graphs increases. The HG provides the best performance on analytics under most conditions, and especially for larger time periods, with a linear increase but a slower rate of growth (Figures 5.23b, 5.25). The reason the HG outperforms the OG is because in the HG smaller messages are exchanged during vertex synchronization between supersteps, and fewer vertex-cuts are made within each snapshot group. Additionally, with multiple snapshot groups, several supersteps can be computed simultaneously, one for each group, which somewhat compensates for the effect of stragglers on the superstep completion.

The tradeoff between the OG and the HG depends on graph evolution characteristics. If the graph is a growth-only evolution (such as in Twitter), then no more than one snapshot group can be formed. In such cases the computation in the HG representation is exactly the same as in the OG representation. The finding that HG still outperforms OG on the Twitter dataset (Figure 5.25) is unexpected and bears further investigation. If the edge evolution represents more transient connections, then the HG is expected to scale better than the OG (Figure 5.23a).

In summary, nonrecursive aggregation is best supported by the VE representation, while the recursive aggregation by the HG representation.

5.4.5 Node Creation

The performance of the attribute-based node creation operator was evaluated over the RG and the VE representations, varying the size of the query temporal window with a trim operator. We group the wiki-talk dataset using the second property, number of edits, that is available for each user.

The attribute-based node creation operator is currently implemented over a simple non-recursive case of matching property values over nodes, which is supported using a group-by operation. Thus

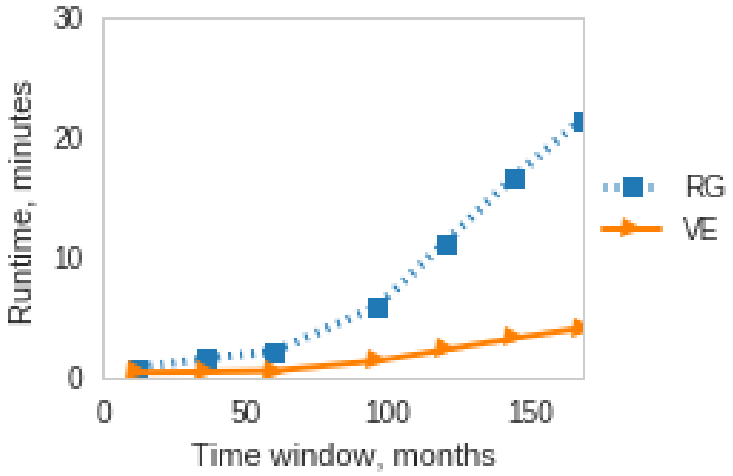


Figure 5.26: Performance of attribute-based node creation on the wiki-talk dataset with varying temporal window, over the edit count node property.

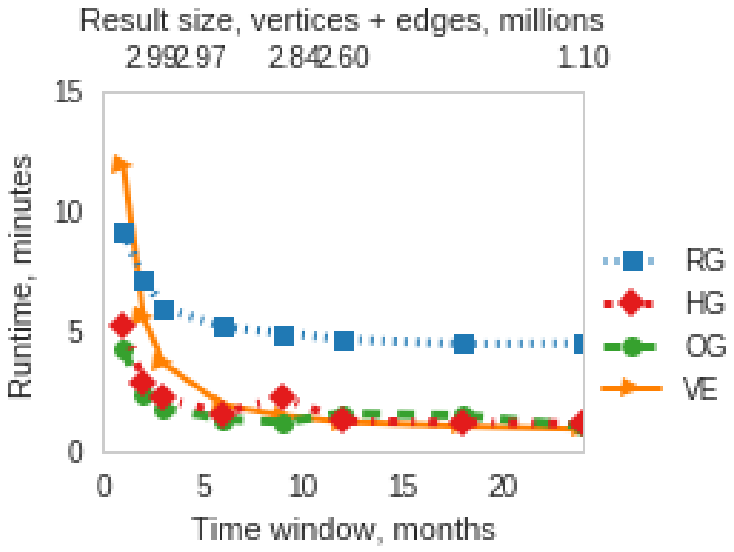
the performance is linear in the size of the input both for VE and RG, but scales significantly worse for RG due to being executed on each snapshot individually, as with all other operations in RG.

Window-based node creation (Def. 3.2.10) performance was evaluated on all representations, since all have different implementations of this operator. We executed topology-only creation (no attributes), varying the size of the temporal window. We observe that performance depends heavily on the quantification, and on the data evolution rate. OG and HG are aggregated data structures with good temporal locality and thus in most cases provides good performance and are not very sensitive to the temporal window size (Figures 5.27a- 5.28b). However, in datasets with a large number of representative graphs (such as nGrams), OG is slow on large windows, a factor of 2 worse than VE in the worst case (Figures 5.28a, 5.28b). VE outperforms OG and HG when vertex and edge quantification levels match (Figure 5.28a), but is worse than OG and HG when vertex quantification is stricter than edge quantification and FK must be enforced (Figure 5.27b). OG and HG also outperform VE when the time window is small. RG does not provide reasonable performance on any of our datasets.

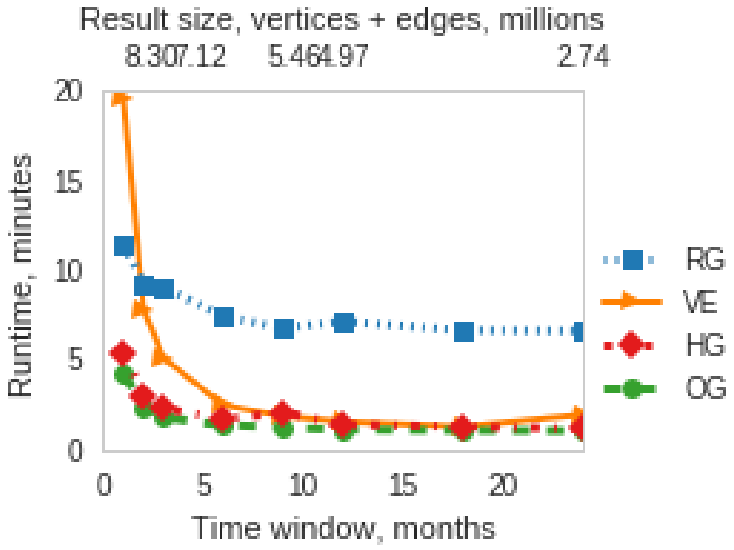
In summary, in most cases, HG provides the best performance on window-based node creation operation. However, VE is likely to lead to better performance when the window size is larger than most node and edge periods of validity.

5.4.6 Binary Operations

Union, intersection, and difference by structure were evaluated by loading two time slices of the same dataset with varying temporal overlap. Performance depends on the size of the overlap (in

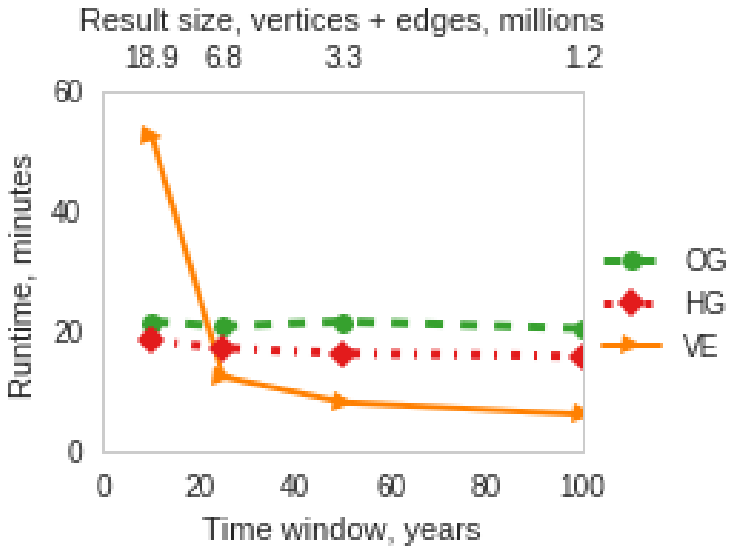


(a) r_v = always, r_e = always

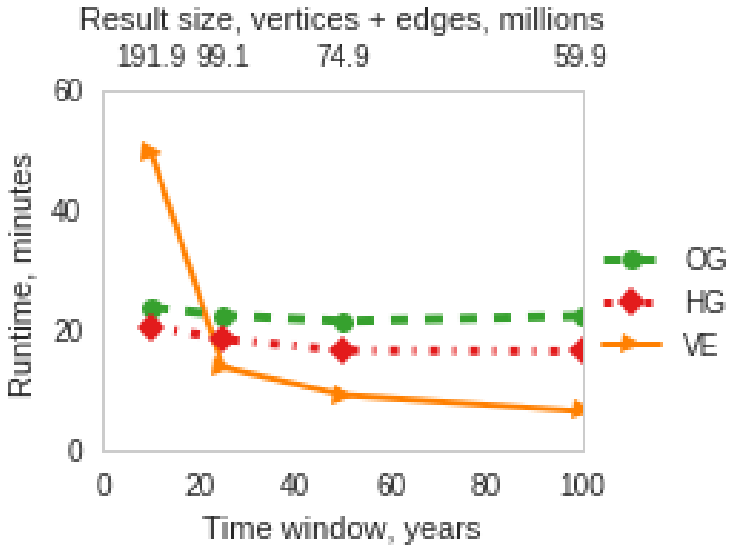


(b) r_v = always, r_e = exists

Figure 5.27: Performance of temporal-window node creation on the wiki-talk dataset with varying window size. $\mathsf{node}_w^T(\mathsf{w} = \mathsf{w}_1, \mathsf{r_v} = \mathsf{always}, 'wiki - talk')$

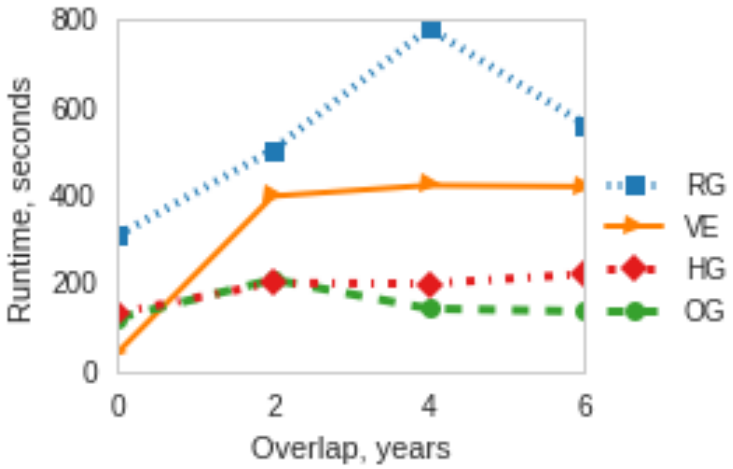


(a) r_v = always, r_e = always

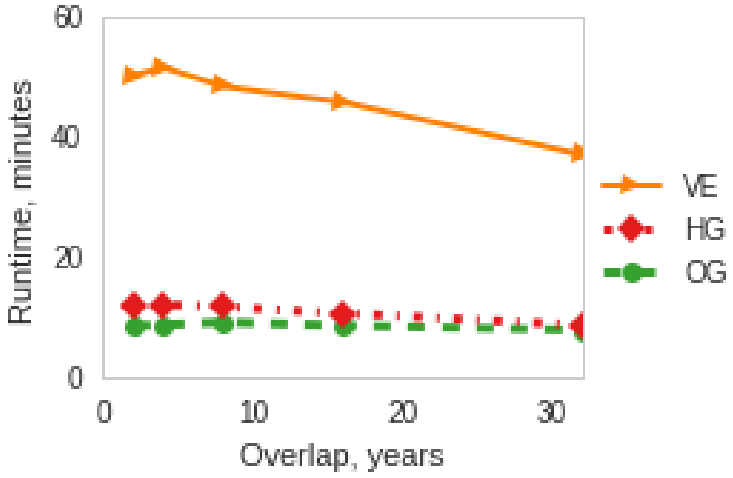


(b) r_v = always, r_e = exists

Figure 5.28: Performance of temporal-window node creation on the nGrams dataset with varying window size. $\mathsf{node}_w^T(\mathsf{w} = \mathsf{w}_1, \mathsf{r_v} = \mathsf{always}, 'nGrams')$

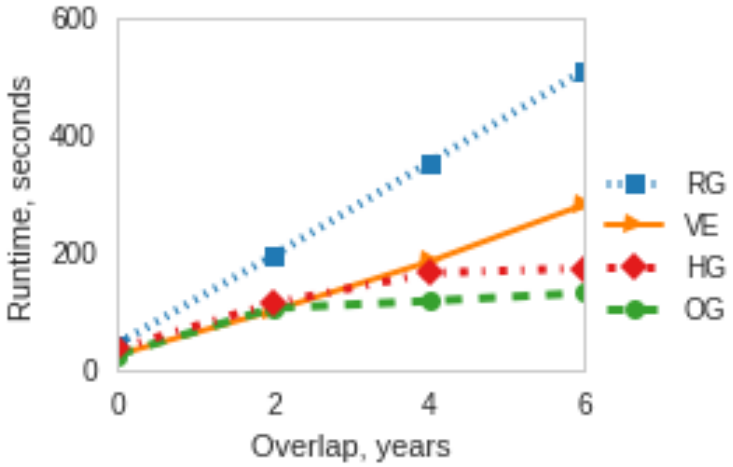


(a) on wiki-talk

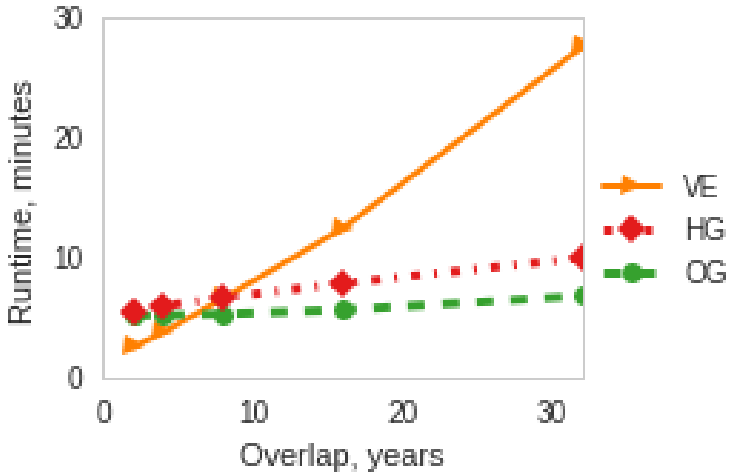


(b) on nGrams

Figure 5.29: Performance of the union operator with varying temporal overlap over the same dataset.

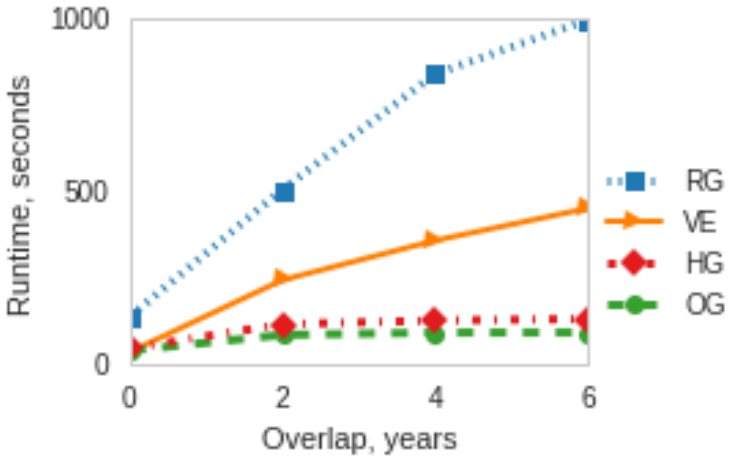


(a) on wiki-talk

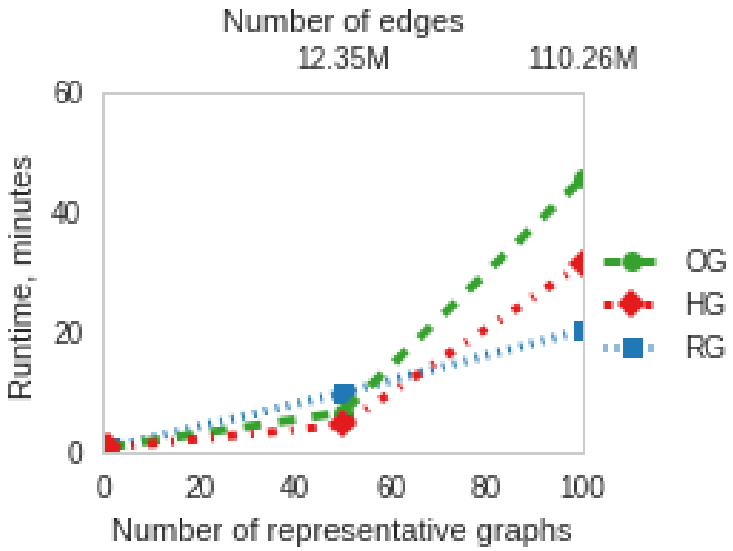


(b) on nGrams

Figure 5.30: Performance of the intersection operator with varying temporal overlap over the same dataset.



(a) on wiki-talk



(b) on nGrams

Figure 5.31: Performance of the difference operator with varying temporal overlap over the same dataset.

the number of representative graphs) and on the evolution rate. OG always has good performance, near-constant with respect to the overlap size (Figures 5.29, 5.30, 5.31). This is expected, since OG union, intersection, and difference are implemented as joins (outer or inner) on the vertices and edges of the two operands. VE, on the other hand, splits the coalesced vertices/edges of each of the two operands into intervals first, takes a union, and then reduces by key. When evolution rate is low and duration of an entity is high, such as in wiki-talk for vertices, the split produces a lot of tuples to then reduce, and performance suffers (Figure 5.29a).

RG only has good performance on intersection when few representative graphs overlap, and never on union (Figure 5.30b).

HG performance is worse than OG, by a constant amount in union, and diverges in intersection.

difference performance is similar to intersection, as can be seen in Figure 5.31.

5.5 Comparison to Baseline

In this Section we describe a series of experiments to compare Portal with two available baselines: the G* system [62] and a simulation of the ImmortalGraph system [74]. Of the three systems in published literature that support queries on evolving graphs, only G* is available publicly. The ImmortalGraph system was developed by Microsoft Research and our request for a copy was declined. The third system, the Historical Graph Store [58], was also not available. For a more in-depth discussion about these systems, see Section 6.5.

Our experiments show that Portal outperforms both baselines on complex queries, in particular for queries over long time ranges.

5.5.1 GStar as a Baseline

The G* system experiments were conducted on the same 16-slave 1-master cluster. G* was installed on all machines and the datasets were converted to the G* format. We used a small DBLP dataset [1] that contains co-authorship information from 1936 through 2016 (80 representative graphs), with about 2.4 million author nodes and over 7 million undirected co-authorship edges. We also used a subset of the wiki-talk dataset from 2002 to 2012 (130 representative graphs). All attempts to use a larger subset of the wiki-talk dataset resulted in the G* system crash and data corruption.

The G* system ingests the evolving graph data one snapshot at a time and replicates them across all machines in an uncompressed state. The DBLP dataset occupies 238M of disk on each

machine (3808M in total), in contrast to 123 in total M on HDFS with no-partition method in Portal. The wiki-talk subset occupies 5GB on each machine in G*, in contrast to 156.5M for the full interval on HDFS with no-partition method in Portal.

We compare Portal performance with G* on the four test queries that were reported in the G* publication ([62], Section 2.3.2, Fig. 4): average vertex degree, distribution of the node clustering coefficient, distribution of the minimum path, and the distribution of the weakly connected component size. We verified that the performance of G* on the four queries in our setup exhibits the same trends as reported by the authors, although one query, query 3, is inconclusive. We implemented each query in Portal using two different approaches: using only TGA operators, and using a mixture of TGA operators and SparkSQL [9] operators. The Portal-SQL method was used for a closer comparison to how G* queries are structured, where graph and relational operators are mixed.

G* outperforms Portal on small data sizes for all four queries, but its performance degrades as the query time range is increased. Portal outperforms G* on large data sizes and on large query time ranges, sometimes by an order of magnitude or more. The only exception is query 3, minimum distance distribution, which should be investigated further.

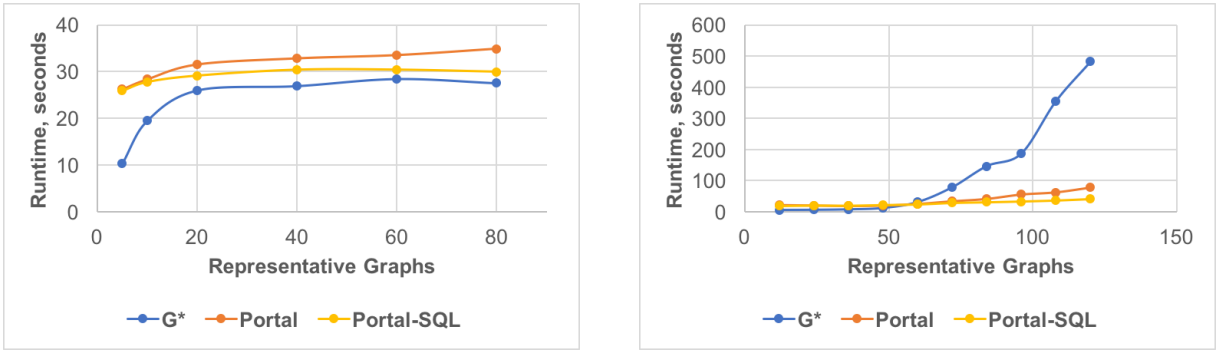
Average Vertex Degree

The average vertex degree test query computes an average degree for each representative graph. Listing 5.1 shows the corresponding G* query in the G* declarative language DGQL. Listing 5.2 shows the same query in Portal. We use the aggregation (Def. 3.2.8) operator to compute verge degree, and then an attribute-based node creation operator (Def. 3.2.9) to summarize each representative graph. Listing 5.3 shows the query using an approach mirrowing the G* approach, i.e., using graph operators, followed by relational operators. We used the VE representation for this query as it showed the best performance on the aggregation queries (Section 5.4.4).

Listing 5.1: G* query for the average vertex degree.

```
select graph.id, avg(degree)
from (select graph.id, degree(vertex) as degree
   from graph('/dblp/range))
group by graph.id
```

Listing 5.2: Portal query for the average vertex degree.



(a) DBLP dataset (b) wiki-talk dataset

Figure 5.32: Performance of G* and Portal on the average node degree query.

```
//load data
val g = GraphLoader.buildVE(data, -1, -1, range)
//compute degree per vertex
val degs = g.aggregateMessages[Int](sendMsg = (et =>
   Iterator((et.dstId,1),(et.srcId,1))), (a,b) => a+b, 0, TripletFields.None)
//compute one vertex per rg with sum of degrees and count of vertices
val rgs = degs.vmap((vid, intv, attr) => (attr._2, 1), (0,0))
   .createAttributeNodes( (a, b) => (a._1+b._1, a._2+b._2))((vid,attr) => 1L)
val result = rgs.vmap((vid, intv, attr) => (attr._1 / attr._2.toDouble), 0.0)
result.vertices
```

Listing 5.3: Portal-SQL query for the average vertex degree.

```
//load data
val g = GraphLoader.buildVE(data, -1, -1, range)
//compute degree per vertex
val degs = g.aggregateMessages[Int](sendMsg = (et =>
   Iterator((et.dstId,1),(et.srcId,1))), (a,b) => a+b, 0, TripletFields.None)
val df = makeDataFrameInt(degs.vmap((vid, intv, attr) => attr._2, 0).vertices)
df.groupBy("estart").agg(avg("attr"))
```

G* outperforms Portal with and without SQL for small ranges of the evolving graph history, and especially in small datasets. For the DBLP dataset, G* is always faster than Portal, although the difference in performance is small for the largest data sizes – 27.5 seconds for G* vs. 30 seconds for Portal-SQL (Figure 5.32a). However, as the data size grows and the range of the query, expressed as the number of representative graphs, grows, Portal outperforms G* by an order of magnitude

– 482 seconds for G* vs. 78 seconds for Portal, for the largest query on the wiki-talk dataset (Figure 5.32b).

Portal does not store the degree information for each node, whereas the G* system stores each edge node with a list of its edges, so degree computation is as simple as taking a list size. This explains why G* outperforms Portal on small number of representative graphs. Note, however, that G* performance degrades drastically with the time range increase, whereas Portal's grows linearly, with a small slope. One possible explanation for such disparity is that, while G* graph operators are distributed, the aggregation step (group by Listing 5.1) appears not to be. It is also possible that G* cannot handle large data sizes, especially when the temporal dimension is extended. We note that the results reported by the authors are primarily either over a small temporal range (up to 12 snapshots) or over a single snapshot.

The combination of Portal for graph operations and SQL for summarization provides the best overall performance, although the difference in total time between pure Portal queries and Portal-SQL queries is small – 34.9 seconds for Portal vs. 30 seconds for Portal-SQL in DBLP and 78 seconds for Portal vs. 40.6 seconds for Portal-SQL for wiki-talk. One possible explanation for the performance improvement with SQL is that SparkSQL stores data in memory in a more efficient manner than pure RDDs do. SparkSQL queries have been shown to be faster than the corresponding RDD transformations [9]. There are two reasons why Portal itself is built on RDDs rather than SparkSQL DataFrames: the functional method to coalescing using a group and fold operation is significantly less complex than a deep nested query with NOT EXISTS that is required to perform coalescing in pure SQL, and 2) RDDs are better suited for storing nonatomic attributes and heterogeneous attributes.

Clustering Coefficient Distribution

The clustering coefficient distribution query computes the distribution of the node clustering coefficient for each representative graph. A clustering coefficient is a measure of how close a node is to being in a clique and is computed as a ratio of the number of triangles that pass through the node to the number of node neighbors [91]. Listing 5.4 shows the corresponding G* query in DGQL. Listing 5.5 shows the same query in Portal. We use the clusteringCoefficient snapshot analytic on the HG representation, since HG provides the best performance on analytics in the general case (Section 5.4.4). To compute the distribution of the coefficients, we use an attribute-based node creation operation, similarly to the average degree query above, but creating a node for each value

of the coefficient, rounded to one decimal point. Finally, Listing 5.6 shows the query using Portal graph operators, followed by relational operators in SparkSQL.

Listing 5.4: G* query for the computation of the clustering coefficient distribution.

```
select graph.id, coeff10*0.1, count(*)
from (select graph.id, floor(c\_coeff(vertex)*10) as coeff10
   from graph('/dblp/range'))
group by graph.id, coeff
```

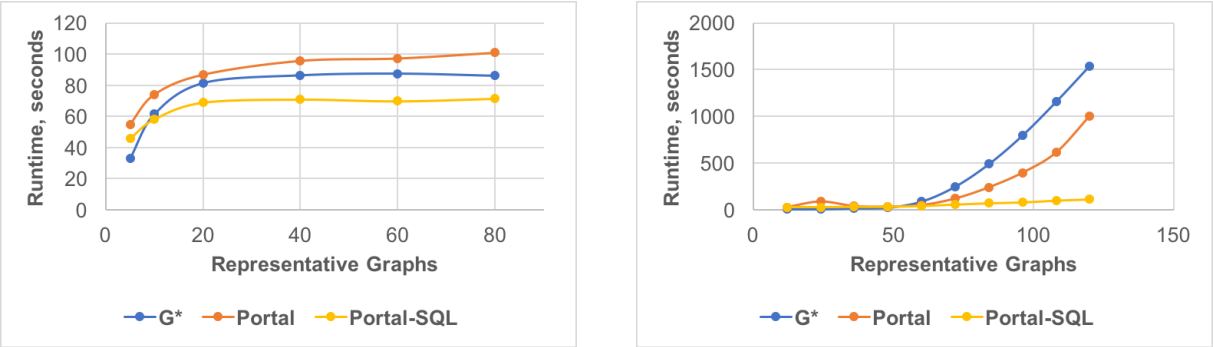
Listing 5.5: Portal query for the clustering coefficient distribution.

```
//load data
val g = GraphLoader.buildHG(data, -1, -1, range)
//compute clustering coefficient per vertex
val coeff = g.clusteringCoefficient()
//compute one vertex per rg per clustering coefficient range
//i.e., 0-0.1, 0.1-0.2, etc.
val distro = coeff.vmap((vid, intv, attr) => (math.floor(attr._2*10)/10, 1),
   (0.0,0))
   .createAttributeNodes( (a,b) => (a._1, a._2 + b._2))
    ((vid,attr) => (attr._1*10).toLong)
distro.vertices
```

Listing 5.6: Portal-SQL query for the clustering coefficient distribution.

```
//load data
val g = GraphLoader.buildHG(data, -1, -1, range)
//compute clustering coefficient per vertex
val coeff = g.clusteringCoefficient()
val df = makeDataFrameDouble(coeff.vmap((vid, intv, attr) =>
   math.floor(attr._2*10)/10, 0.0).vertices)
df.groupBy("estart", "attr").agg(count("vid"))
```

The results on query 2 exhibit similar trends as in query 1. G* outperforms Portal for small data sizes, but its performance degrades as the query time range is increased. In fact, even for the small DBLP dataset Portal-SQL outperforms G* on all but the smallest time ranges (Figure 5.33a). For the largest range, G* returns a result in 86 seconds vs. 71.4 seconds for Portal-SQL.



(a) DBLP dataset

(b) wiki-talk dataset

Figure 5.33: Performance of G* and Portal on the clustering coefficient distribution query.

As with query 1, Portal outperform G* for all larger date range queries on the wiki-talk dataset (Figure 5.33b) – 25.6 minutes for G* on the largest range vs. 16.7 minutes for Portal and under 2 minutes for Portal-SQL.

The best performance overall is provided by the combination of Portal with SQL. In fact, the difference is substantial – 8x. This indicates that SparkSQL aggregation (group by) is significantly more efficient than a similar operation on RDDs.

Minimum Distance Distribution

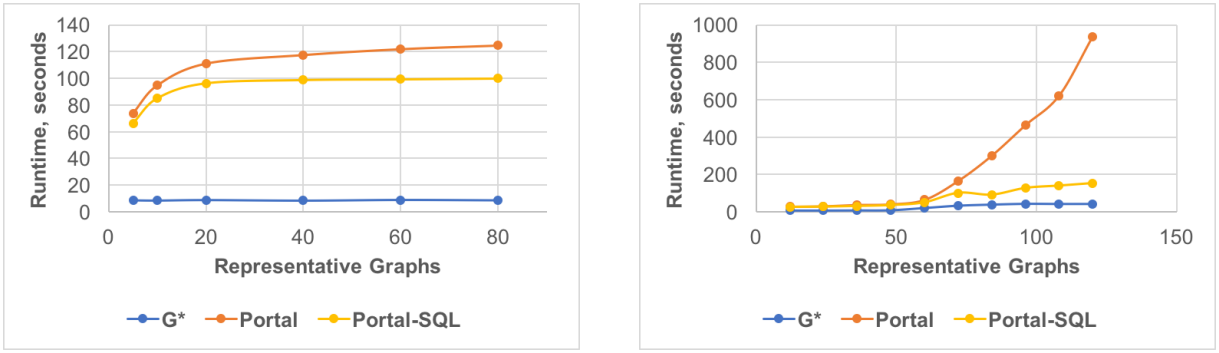
The minimum distance distribution query computes the distribution of the shortest distance to the specified node from all other nodes, for each representative graph. Listing 5.7 shows the corresponding G* query in DGQL. Listing 5.8 shows the same query in Portal. We use the shortedPaths analytic, followed by attribute-based node creation, similarly to query 2. Finally, Listing 5.9 shows the query using the Portal graph operators, followed by relational operators in SparkSQL.

Listing 5.7: G* query for the computation of the minimum distance distribution.

```
select graph.id, min\_dist, count(*)
from min\_dist(graph('/dblp/range'), '1')
group by graph.id, min\_dist
```

Listing 5.8: Portal query for the minimum distance distribution.

```
//load data
val g = GraphLoader.buildHG(data, -1, -1, range)
//compute shortest path
val sp = g.shortestPaths(true, Seq(src))
//compute one vertex per rg per distance
```



(a) DBLP dataset

(b) wiki-talk dataset

Figure 5.34: Performance of G* and Portal on the minimum distance distribution query.

```
val distro = sp.vmap((vid, intv, attr) =>
   (attr.\_2.getOrDefault(src, Int.MaxValue), 1), (Int.MaxValue,0))
   .createAttributeNodes( (a,b) => (a._1, a._2+b._2))((vid,attr) => attr._1.toLong)
distro.vertices
```

Listing 5.9: Portal-SQL query for the minimum distance distribution.

```
//load data
val g = GraphLoader.buildHG(data, -1, -1, range)
//compute shortest path
val sp = g.shortestPaths(true, Seq(src))
val df = makeDataFrameInt(sp.vmap((vid, intv, attr) =>
   attr._2.getOrDefault(src, Int.MaxValue), Int.MaxValue).allVertices)
df.groupBy("estart", "attr").agg(count("vid"))
```

The results of the minimum distance query are not in line with all other results. G* exhibits a constant time performance in DBLP – about 8.6 seconds for every date range (Figure 5.34a, and a linear time performance in wiki-talk – 40.6 seconds in the largest case (Figure 5.34b). In constrast, Portal time grows as a function of graph size rather than a function of the number of representative graphs.

There are two possible explanations for the large disparity in performance between G* and Portal on this query. The method to compute shortest paths may be different, as G* stores edges with their corresponding nodes, whereas Portal uses a vertex-cut approach (Section 4.1). Another possible explanation is a difference in default behavior for nodes that do not have a path to the target node. In Portal every node reports a a MAXINT distance if there is no path. In G* no distance is reported.

Finally, as with query 2, this, in combination with the performance of Portal-SQL, gives further indication that the implementation of the attribute-based node creation operator should be investigated.

It is worth noting that the results we obtained for G* on this query do not closely resemble those that are reported in the G* paper, where query 3 exhibits the slowest performance of all four queries. However, query 3 was evaluated on a synthetic tree graph in the G* paper, whereas both the DBLP and the wiki-talk dataset are sparse.

Component Size Distribution

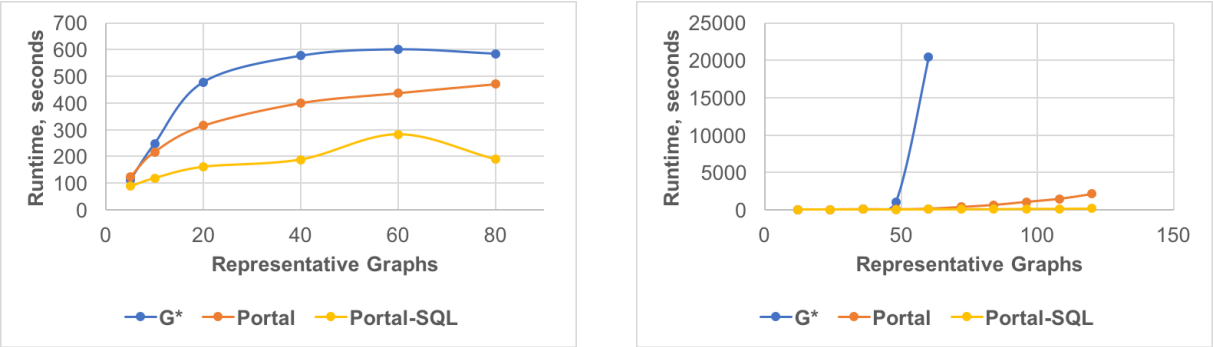
The final query, component size distribution, computes the distribution of weakly connected component sizes for each representative graph. Listing 5.10 shows the corresponding G* query in DGQL. Listing 5.11 shows the same query in Portal. We use the connectedComponents analytic, followed by two invocations of the attribute-based node creation – one to group by the component, and the second to compute the size distribution. Finally, Listing 5.12 shows the query using the mixture of Portal and SparkSQL.

Listing 5.10: G* query for the computation of the connected component size distribution.

```
select graph.id, comp\_size, count(*)
from (select graph.id, comp\_id, count(*) as comp\_size
   from comp\_id(graph('/dblp/range'))
   group by graph.id, comp\_id)
group by graph.id, comp\_size
```

Listing 5.11: Portal query for the connected component size distribution.

```
//load data
val g = GraphLoader.buildHG(data, -1, -1, range)
//compute connected components
val ccs = g.connectedComponents()
//compute one vertex per component
val comps = ccs.vmap((vid, intv, attr) => (attr._2, 1), (0L,0))
   .createAttributeNodes( (a,b) => (a._1, a._2+b._2))((vid,attr) => attr._1)
//compute one vertex per component size for distribution
val distro = comps.vmap((vid, intv, attr) => (attr._2, 0), (0,0))
   .createAttributeNodes( (a,b) => (a._1, a._2+b._2))((vid,attr) => attr._1)
```



(a) DBLP dataset

(b) wiki-talk dataset

Figure 5.35: Performance of G* and Portal on the connected component size distribution query.

```
distro.vertices
```

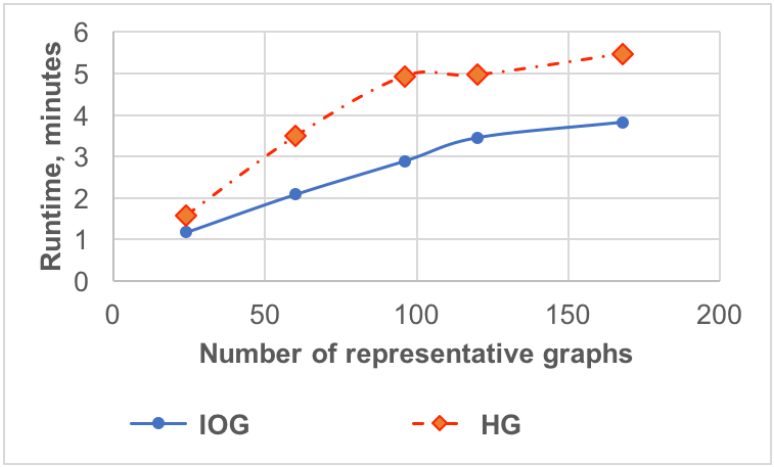
Listing 5.12: Portal-SQL query for the connected component size distribution.

```
//load data
val g = GraphLoader.buildHG(data, -1, -1, range)
//compute connected components
val ccs = g.connectedComponents()
val df = makeDataFrameLong(ccs.vmap((vid, intv, attr) =>
   attr._2, -1L).vertices)
df.groupBy("estart", "attr").agg(count("vid").as("comp_size"))
  .groupBy("estart","comp_size").agg(count("attr"))
```

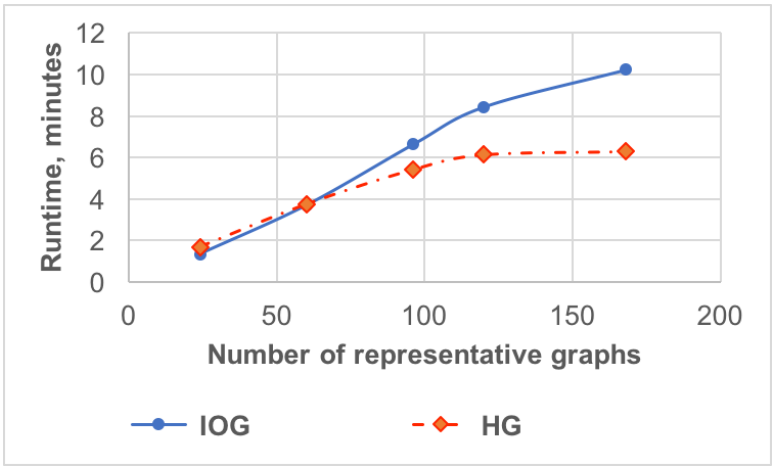
Portal outperforms G* on this query for all date range sizes in both datasets. On the largest data size in DBLP G* completes in 584.4 seconds vs. 189.3 for Portal-SQL, a 3x difference (Figure 5.35a). The difference is even more significant with the wiki-talk dataset. G* did not complete a query within 6 hours on any of the second half of the queries, whereas Portal exhibits linear increase in performance as a function of data size (Figure 5.35b). Portal completes the largest query in the wiki-talk dataset in about 34 minutes and Portal-SQL in under 3 minutes.

5.5.2 ImmortalGraph as a Baseline

The ImmortalGraph system is the first published evolving graph system [45, 74]. The exact range of its capabilities is unclear because the API is not discussed in detail and the system itself is not available to the public due to proprietary rights concerns. The main operation that



(a) connected components



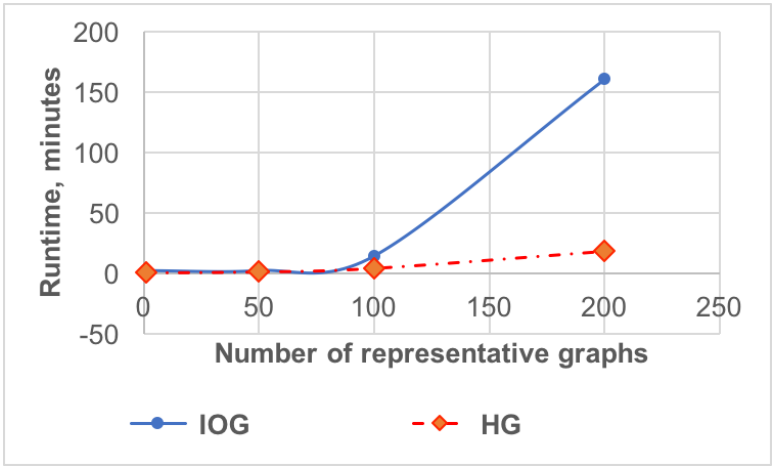
(b) PageRank

Figure 5.36: Performance of ImmortalOneGraph (IOG) on snapshot analytics on the wiki-talk dataset.

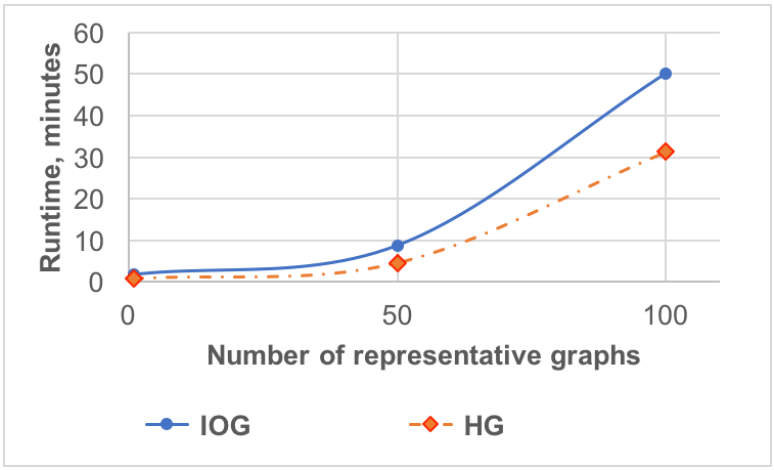
ImmortalGraph supports efficiently is snapshot analytics, using a batching method. (See a more in-depth discussion in Section 6.5.)

In order to compare Portal performance with ImmortalGraph, we implemented an in-memory representation based on the time-locality in-memory layout. Recall that with time locality, data is grouped such that all evolution for a single node or edge is stored together. The resulting data structure, ImmortalOneGraph, is very similar to our OG representation. However, OG only stores bit sets of node and edge validity, while the attributes are stored separately (Section 4.4.3), whereas ImmortalOneGraph stores the attributes with the node in a single GraphX graph.

ImmmortalOneGraph (IOG) uses a batching method for snapshot analytics, computing node values for all times simultaneously. We evaluate the performance of the IOG on the Weakly Connected Components and the PageRank snapshot analytics.

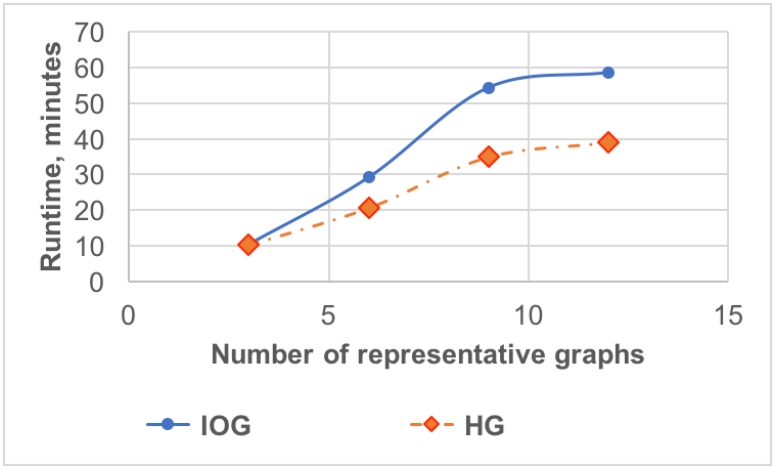


(a) connected components

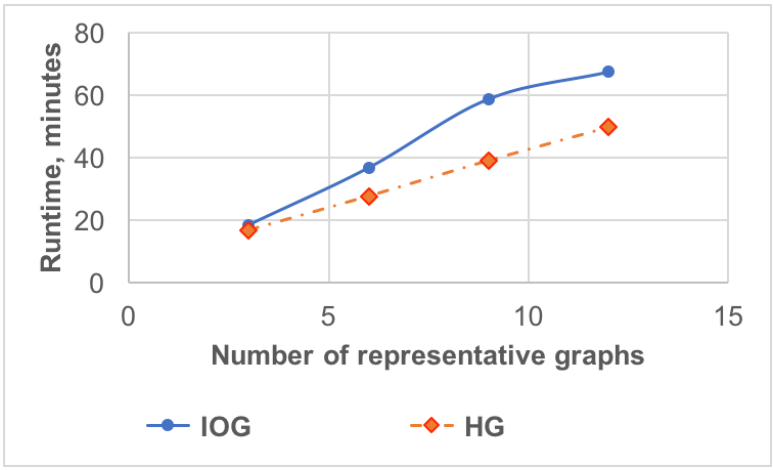


(b) PageRank

Figure 5.37: Performance of ImmortalOneGraph (IOG) on snapshot analytics on the nGrams dataset.



(a) connected components



(b) PageRank

Figure 5.38: Performance of ImmortalOneGraph (IOG) on snapshot analytics on the Twitter dataset.

The simulated IOG outperforms Portal's HG on the connected components analytics on the wiki-talk dataset (Figure 5.36a), but underperforms on the more complex PageRank analytic (Figure 5.36b). The explanation for this difference lies in the fact that the connected components algorithm converges quickly on a sparse graph such as wiki-talk, whereas PageRank computes a new value for the vast majority of nodes in each superstep. As a result, the number of messages exchanged while executing PageRank is much higher than while executing connected components, and the overall communication cost higher. HG has a better combination of structural and temporal locality that reduces communication costs in such scenarios, as we discussed in Section 5.4.4.

On the larger datasets nGrams and Twitter HG outperforms IOG in all cases, as can be seen in Figures 5.37 and 5.38. We have to be careful to interpret these results because we are simulating the performance of ImmortalGraph without access to the original code and in a different distributed environment than reported in the original paper [74]. However, there is strong evidence to indicate that a hybrid layout provides a better locality in most cases.

5.6 Usecases

To see how Portal handles the use cases from Section 1.3, we implemented each one over the wiki-talk dataset. Each example requires a sequence of operators. For each operator we used the best performing data structures based on the comparison experiments described above.

Example 1 answers the question of whether there are high influence nodes and whether that behavior is persistent in time. The Scala code to compute the answer is displayed in Listing 5.13 and the query took 76 seconds to execute. The results show that from 25 nodes with mean degree of 40 and above that have persisted for at least 6 months, 6 have coefficient of variation below 50, which is quite low, and only 5 have it above 100. This indicates that there are in fact high in-degree nodes and that they continue to be influential over long periods of time, despite the loose connectivity of the overall network.

Listing 5.13: Portal program to compute high influence nodes.

```
val deg = g.aggregateMessages[Int](triplet =>
    { Iterator((triplet.dstId, 1)) }, (a,b) => {a+b}, 0, TripletFields.None)
   .vmap((vid, intv, attr) => Map(intv -> attr._2), Map[Interval,Int]())
val spec = ChangeSpec(g.getTemporalSequence.count.toInt)
val agg = deg.createTemporalNodes(spec, Exists(), Exists(),
   (a, b) => {a ++ b}, anyfun)
```

Table 5.2: In-degree centrality over time in wiki-talk

Window	Mean	StDev
1	0.03	0.09
2	0.04	0.11
3	0.04	0.13
6	0.06	0.18
12	0.03	0.05

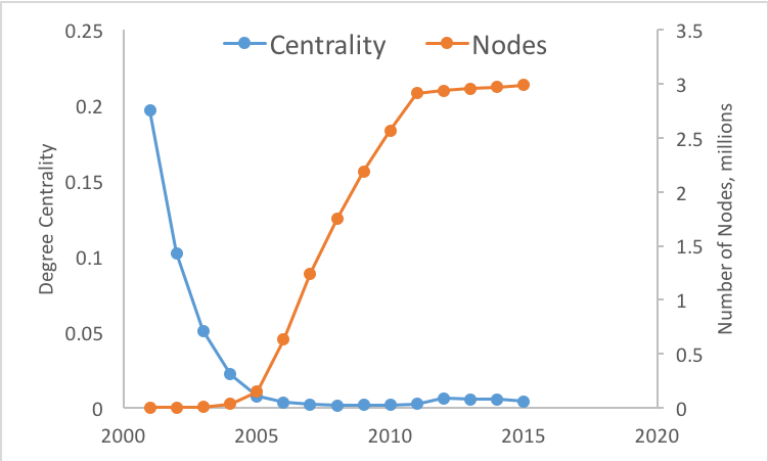


Figure 5.39: In-degree centrality with 1 year resolution.

```
val unit = Resolution.from("P1M").unit
println("lowest coefficient of variation top 10 users: " +
   agg.vmap((vid, intv, attr) => {
      val allpoints = TempGraphOps.makeSeries[Int](attr, Some(0))
        .map(x => x.getOrElse(0));
      val mean = allpoints.sum / allpoints.size.toDouble;
      val variance = allpoints.map(x => math.pow(x - mean, 2))
        .reduce(_ + _) / allpoints.size;
      (mean, math.sqrt(variance) / mean * 100, allpoints.size)
   }, (0.0, 100.0, 1))
   .vertices.filter(x => x._2._2._1 > 0.0)
   .sortBy(x => x._2._2._1, ascending = false).take(50).mkString("\n"))
```

Example 2 examines how the graph centrality changes over time. The program is 6 lines of Scala code iterating with temporal windows of 1, 2, 3, 6, and 12 months, and the analysis took 25 minutes. Results show that regardless of the temporal resolution, the in-degree centrality is extremely low, about 0.04. Figure 5.39 provides an explanation – as the size of the graph increases, its centrality decreases. Given that the number of edges in this graph is only about 4 times the number of nodes, the graph is too sparse and disjointed to have any centrality.

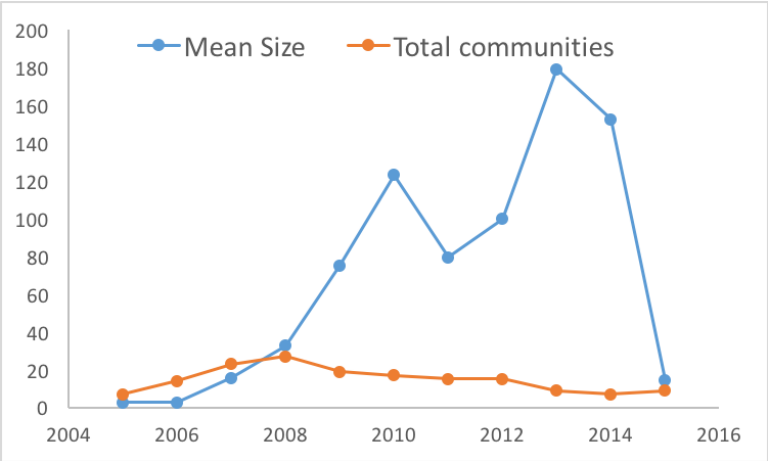


Figure 5.40: Communities with 1 year resolution.

Finally, example 3 examines whether communities can be detected in the wiki-talk network at different temporal resolution. The program, similar to the one above, is 6 lines of Scala code with varied temporal windows. The total runtime is 58 minutes. Communities, defined as connected components, can be detected in all temporal resolutions. As a reminder, the edge quantification in this query is always, so only edges that persist over each window are retained. The presence of communities even with large temporal resolution indicates that communities form and persist over time. Figure 5.40 shows the mean size of all communities by time and their total number. The peaks of the mean size, visible in all temporal windows, may indicate that communities form and then reform in a different configuration, perhaps for a different purpose. The results of this analysis can serve as a starting point to investigate the large communities and what caused the size shifts.

In summary, complex analyses can be expressed as queries in Portal and lead to interesting insights about the evolution of the underlying phenomena.

Chapter 6: Related Work

In this chapter we place our work in the context of published work on evolving graph models (Section 6.1), evolving graph queries (Sections 6.2 and 6.3), and systems 6.5.

6.1 Evolving Graph Models

While the temporal models in the relational literature are very mature, the same cannot be said for the evolving graphs literature. Evolving graph models differ in what time model they adopt (point or interval), what top-level entities they model (graphs or sets of nodes and edges), whether they represent topology only or attributes or weights as well, and what types of evolution are allowed. All evolving graph models require node identity, and thus edge identity as well, to persist across time.

The first mention of evolving graphs that we are aware of is by Harary and Gupta [46] who informally proposed to model the evolution as a sequence of static graphs, as depicted in Figure 6.1. This model has been predominant in the research literature since [18, 22, 36, 38, 55, 57, 63, 65, 77, 81, 84, 94], with various restrictions on the kinds of changes that can take place in the graph evolution. For example, Khurana and Deshpande use this model but a node, once removed, cannot reappear [57]. In [36, 62, 77] there is no notion of time at all, only a sequence of graphs. In [18] and [55] the time series of graphs represent topology only, with no attributes, and only edges can vary with time, while the nodes remain unchanged. Ferreira's model [38] allows both node and edge evolution, but again, only restricted to topology.

The advantages of the snapshot sequence model are that it is simple and if snapshots are obtained by periodic sampling, which is a very common approach, it accurately represents the state at that point without making statements about other unknowable times. For example, the WWW is so large that it is impossible to create a fully accurate snapshot that represents any moment in time.

The snapshot sequence approach has several limitations. The state of the graph can be undefined at some time t unless a snapshot is associated with each possible value in the discrete range

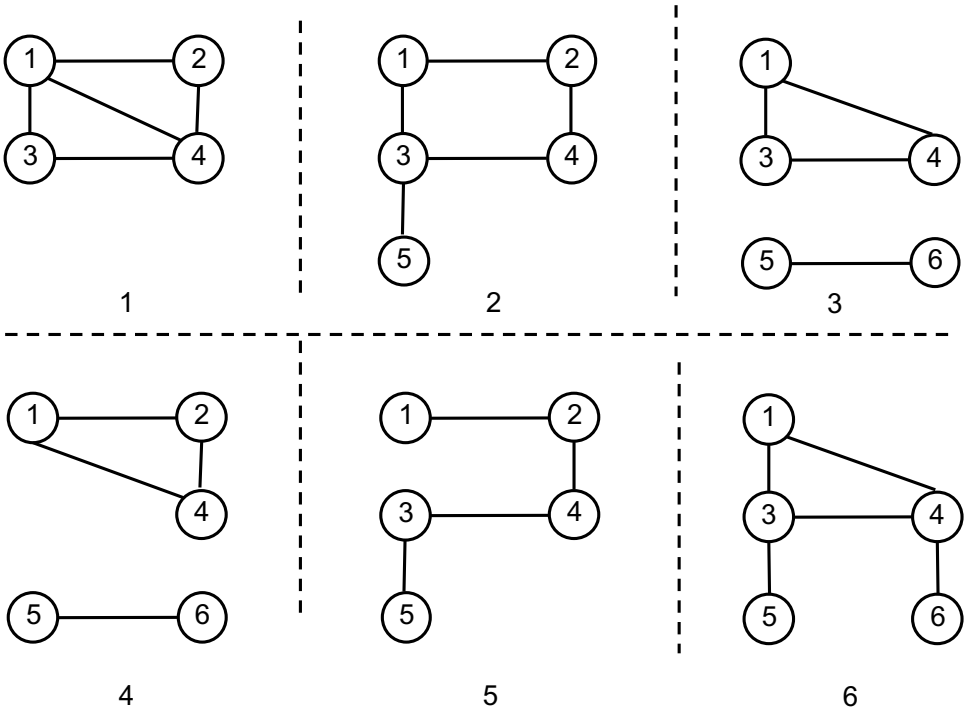


Figure 6.1: Example snapshot sequence representing graph evolution over six stages.

$[t_{start}, now)$. So either there are many consecutive equivalent snapshots or the state of the graph is undefined. It also forces a specific time granularity since the time is discrete, whereas open-closed time intervals can be broken down into any desired level of granularity.

The biggest problem with the snapshot sequence as the underlying conceptual model is that it restricts the range of operations that can be naturally expressed. For example, it is impossible to select a subgraph over a snapshot sequence using a temporal predicate, as we do in Definition 3.2.7, because each snapshot is nontemporal and independent of the others. Stated formally, the snapshot sequence model lacks the extended snapshot reducibility property.

It is worth noting that while the logical model adopted by many researchers is that of a snapshot sequence, the representations differ widely. Huo and Tsotras model an evolving graph as a set of nodes and a set of edges where each entity is interval-stamped [51]. However, nodes have no attributes in this model and the edge attributes are weights. Whether this model is coalesced or not is not formally stated, but we must assume it is uncoalesced. Similary, Koloniari [60], while not providing a formal model, represents nodes and edges as having periods of validity.

Another approach, for labeled directed graphs, is to add annotations to the nodes and edges, corresponding to the changes made. The DOEM database model annotates nodes and arcs (edges) with additions, removals, and updates, with corresponding times at which the change occurred [24]. Multiple annotations may be associated with a single entity. The resulting graph is similar to

Table 6.1: Classification of temporal graph queries, according to Miao et al. [74]

	Point	Range
Global	"What is the diameter of the graph at time t?"	"How did PageRank evolve in the last 1 month?"
Local	"How many friends do I have at time t?"	"Who are my friends since 2011?"

maintaining a single graph with interval timestamps (as our OneGraph), where the changes are stored as deltas rather than materialized as intervals.

The most general model we found is by Casteigts et al. [20] who define a *time-varying graph* as $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$, where V and E are defined as in regular labeled graphs, $\mathcal{T}$ is a time span of the graph from a temporal domain T, ρ is a presence function that maps edges to time instances where they are present, and ζ is a latency function that indicates the time it takes to cross a given edge starting at some time instant. The inclusion of the latency function allows the time-varying graph model to cover not only evolving graphs as we define them, but also temporal transportation and communication networks. The edge labels can be used in place of ids or attributes with no distinction in the model. The nodes in this model do not have labels or attributes, but these could be easily added. Similarly, as the authors note, a node presence function and node latency function can also be added.

A sequence of snapshots can be computed from our model with a slice over all time points. We emphasize that the choice of the logical model impacts the expressiveness of the algebra and advocate our model over the snapshot sequence from that perspective.

Our model supports fully evolved graphs (both topological and attribute evolution for both nodes and edges), uses time intervals for time granularity, and provides both graph-centric and entity-centric views.

In summary, a snapshot sequence limits the range of expressions that can be formed over it. It is one of the contributions of this work to propose a model that lifts this restriction.

6.2 Evolving Graph Queries

Evolving graph analysis and mining have been receiving increased attention over the past few years. Several researchers have proposed individual queries, or classes of queries, for evolving graphs, but without a unifying syntax or general framework.

Miao et al. [74] proposed a taxonomy of evolving graph queries that divides them along two dimenstions: local vs. global and point vs. interval, as shown in Table 6.1. Point queries are

limited to one specific time instant in the evolving graph history. Range queries are over the whole history or a specific subset expressed as an interval. It is useful to think of the term "range" here loosely as it is sometimes desirable to express time constraints more that are more complex than a single interval. For example, to analyze all graph history in election years or only periods of activity longer than a month.

Local queries are those confined to a specific region of the graph, such as a specific node, edge, route, or neighborhood. Global queries operate over the whole graph. These are often called **analytics** in the literature.

The proposed operators can also be divided into temporal, i.e., those that return a temporal result, and nontemporal.

6.2.1 Temporal operators

Temporal operators include retrieval of version data for a particular node and edge [41], of journeys [20, 40], subgraph by time or attributes [51, 58], snapshot analytics [58, 62, 74], and computation of time-varying versions of whole-graph analytics like maximal time-connected component [38] and dynamic graph centrality [65].

Retrieval

Many researchers have focused their work on how to retrieve a subset of the evolving graph based on temporal and/or structural constraints. Temporal constraints can be expressed as a set of time points or an interval. Structural constraints can be specific node or edge identities, paths, or neighborhoods. George and Shekhar propose retrieval of time series for a particular node, edge, or route starting at a particular time or across all evolution history [41]. Huo and Tsotras define sub-TEG, a set of graph snapshots during input time interval [51]. Khurana and Deshpande support snapshot retrieval over a list of points, an interval, or a temporal predicate [57]. In a follow-up paper they call this a timeslice operation and also add a getversions operation that retrieves the evolution of input nodes [58]. The G* system provides a method to retrieve a set of snapshots using either a range expession or a list expression [62].

Clearly there is an agreement that it is necessary to be able to limit analysis only to periods or time instances of interest from the overall graph history. TGA provides both a trim operation and a (vertex- and edge-) subgraph operation with temporal predicates. The combination of the two can be used to compute any subset with temporal and structural constraints.

Whole-graph and recursive computation

As we have shown in Section 2.2.2, whole-graph computations can be expressed as aggregation queries and this type of analysis is very popular. In a distributed setting these queries are usually supported using the Bulk Synchronous Parallel execution model [89] implementations such as Pregel [71]. This carries over to the evolving graph literature. The two main types of whole-graph and recursive computation queries are **snapshot analytics** and **over-time analytics** (terminology is ours).

Snapshot analytics are aggregation queries such as PageRank, weakly connected components, and shortest paths that are semantically carried out over each individual snapshot. This includes reachability queries by Fard [36], temporal iterative graph mining in the Chronos/ImmortalGraph system (to include single-source shortest path, PageRank, maximal independent set, and weakly connected component) [74], clustering coefficient in the Historical Graph Store system [58], several graph operators in the G* system (to include ClusteringCoefficientOperator, SingleSourceShortest-DistanceOperator, PageRankOperator, and WeakComponentOperator) [62], and others. Snapshot analytics are a natural extension of the work done for static graphs, and the focus is primarily on how to carry out the computation efficiently. We describe some approaches in Chapter 4.

Over-time analytics are more complex as they compute patterns *over time*. Casteigts et al. define journeys as a time-extended notion of a graph path that can be used for different kinds of analysis [20]. Ferraira defines several different types of journeys, including foremost and fastest [38]. Foremost journeys compute journeys with the earliest arrival time, while fastest with a minimum delay once the journey starts. He also defines maximal time-connected component, which is a generalization of the maximal connected component to include journeys instead of paths. George et al. use conditional journeys, i.e., journeys with additional temporal predicates, for growing hotspot detection in sensor networks [40]. Huo and Tsotras propose an approach for computing a time-interval shortest path, which generalizes the shortest path computation to the set of distances during a time interval [51]. Lerman et al. propose a new centrality metric for each node, defined as a dynamic measure over a period of time which is an indication of how connected the nodes are over time [65].

Our temporal aggregation (Def. 3.2.8) operation supports both snapshot analytics and over-time analytics, as we show in Section 3.2.4.

Subgraphs and patterns

Looking for patterns within a graph is the most common graph query type, as we already saw in Section 2.2.2. Several researchers have extended subgraph matching into the temporal domain.

Cattuto et al. model a temporal communication network as a set of frames in the Neo4j database and then answer subgraph queries such as common neighborhoods of two users and neighborhood of a target user within a specific time frame [21]. Kan et al. propose querying of evolving graphs for spatio-temporal patterns, looking for correlated subgraphs where the query input is the temporal pattern represented by a waveform [55]. A waveform is a string consisting of 0s and 1s representing edge insertion and deletion events. The matches do not have to be exact to the waveform but within the input temporal distance. Besides the waveform the user can also supply a predicate on a subgraph that can be used to, for example, impose a floor (or ceiling) on the size of the subgraph, or limit it to a certain time interval, or make sure it is connected or a clique, etc. The operator is termed selection, takes in an evolving graph and a query, where query has a waveform and predicate, and returns a set of evolving subgraphs correlated with the waveform that satisfy the predicate.

George et al. propose anomaly detection in sensor networks that can be thought of as subgraph matching based on a computed domain model [40]. The approach selects those nodes whose time series display a pattern of changes inconsistent with the expected model. Jin et al. mine for frequent temporal patterns, where a trend can be an increasing or decreasing node weight. The Historical Graph Store system provides a selection operator that computes entity-centric filtering with a non-temporal predicate [58]. This is essentially just a regular conjunctive query without recursion or time. Lahiri provides an approach for identifying all closed periodic subgraphs, where closed means the subgraph is maximal during some number of time instances [63]. Periodic means that the difference between instances where the subgraph is present is constant. Lionakis et al. describe a system called SQTime that answers social search queries, and specifically node-centric queries with time predicates such as “all friends of x in 2008” [69].

What is common among all these queries is that they either look for a temporal pattern in a time series, which is not inherently graph-centric but clearly useful, or look for subgraphs that exhibit a particular spatio-temporal pattern. TGA supports temporal pattern in a time series with the time-window aggregation operation (Def. 3.2.10) and the matching of spatio-temporal patterns with the temporal subgraph operation (Defs. 3.2.7).

Miscellaneous

In addition to queries in the above categories, several other operators over evolving graphs have been proposed. They are clearly inspired by or directly based on relational algebra operations. For example, the Historical Graph Store system provides a nodecompute operator that takes in an arbitrary function and applies it to each node like map in the MapReduce paradigm [58]. It also provides a tempaggregation operator that applies aggregators, i.e., peak, min, max, mean, over the node time series. The G* system provides a collection of non-graph operators, including AggregateOperator, UnionOperator, JoinOperator, ProjectionOperator, SortOperator, and TopKOperator, that all apply standard relational algebra operators to the outputs of other operators.

6.2.2 Nontemporal operators

Nontemporal operators are focused on local and global point queries. George and Shekhar support retrieval of specific nodes, edges, or routes at a specific time point [41]. The same functionality is also supported in the Historical Graph Store [58] and in Koloniari et al. [60]. Georte et al. support anomaly detection through a subgraph query to locate specific nodes that have anomalous readings (compared to the domain model) at any time point [40]. Khurana and Deshpande provide a snapshot retrieval operation for a specific time point [57], as does the G* system [62]. The main focus in all these approaches is the physical layout of the data and the access methods for fastest retrieval.

Chawathe et al. define a language called Chorel to query an evolving graph model DOEM (described in Section 6.1) representing changes to semistructured data over time [25]. The queries in Chorel allow the user to traverse the graph representing the most recent version of the data or any desired past snapshot, as well as make at time conditions on the change annotations. The Chorel queries correspond to subgraph queries with temporal predicates, where the result is not another graph but rather the set of matches.

Note that any closed algebra on evolving graphs cannot support these operators directly because the result is inherently nontemporal. However, the spirit of the queries can be supported through subgraphs with a temporal predicate or with trim.

Table 6.2: Mapping between published query types and TGA operators.

Query Type	TGA operator
range retrieval	slice, subgraph with temporal predicates
local range retrieval (node, edge, route, neighborhood)	subgraph
snapshot analytics	temporal aggregation
over-time analytics	temporal aggregation
spatio-temporal patterns	subgraph, attribute-based node creation
time series patterns	window-based node creation
node transformation (projection, nodecompute)	v-map
join	intersection

6.3 Mapping

To place our work in context, this section provides mappings between TGA operators and operators in other published works. Table 6.2 lists every type of evolving graph operation that we reviewed in Section 6.2 and showing how it is supported in TGA. We contend that every published evolving graph operation is supported in TGA if it can be expressed in a closed form, i.e., to return another evolving graph. This is an informal statement of coverage. In the next Section we show that TGA also provides operations that are not available in other systems.

6.4 Nontemporal Graph Languages

The TGA is not based on any one specific nontemporal graph language, but rather is an extension of a generalized graph query language based on surveys 2.2.2. In this section we select a subset of the nontemporal graph query languages that have a closed semantics, i.e., that take one or more graphs as input and return a graph as output, and show which TGA operators provide a corresponding temporal operation. These languages are StruQL [37], SocialScope [6], and GraphLog [31]. The semantics of a GraphLog query is not, strictly speaking, closed, but can be interpreted in such a way as to lead to a closed language.

In summary, every TGA operation except edge-map 3.2.5 has an operation in one of these three languages for which it serves as a temporal extension. Another TGA operator that has no nontemporal correspondence is the window-based node creation operator, which is to be expected since it is an inherently temporal operator.

6.4.1 StruQL

The StruQL query language was designed for web-site management within the Strudel system [37]. It operates on labeled graphs rather than property graphs and provides a syntax with a where and collect clauses that correspond to the subgraph operation 2.2.4 and create clauses that corresponds to the node creation operation with a Skolem function to create the object ids 2.2.10. The syntax also allows to combine multiple graphs, with the semantics equal to node and edge set union. All nodes and edge ids that are the result of the query are generated by the Skolem function.

The following example StruQL query produces a site graph that contains all nodes except image files [37]:

```
where   Root(p), p -> * -> q, q -> l -> q', not(isImageFile(q'))
create  N(p), N(q), N(q')
link    N(q) -> l -> N(q'),
collect TextOnlyRoot(N(p))
```

The corresponding TGA operators are the temporal subgraph 3.2.7, temporal node creation 3.2.9, and the temporal union 3.2.11. In the case of the temporal union, no aggregation functions are required because a labeled graph has no properties to resolve.

6.4.2 SocialScope

The SocialScope query language is targeted specifically for querying of social networks [6]. It uses a property graph model with a required type property, similar to the definition we use in this work 2.2.1. SocialScope contains a number of operations, not all of which have a direct correspondence in TGA, but every SocialScope operation can be expressed through some sequence of TGA operators. The Node Selection and Link Selection operators take a condition on the nodes, resp. edges, and an optional scoring function, and output the set of nodes, resp. edges and subset of nodes that are connected by the edges, that meet the condition. The Node Selection operation corresponds to the nontemporal subgraph operation 2.2.4 with a basic graph pattern 2.2.2 with no edges in the pattern. The Edge Selection operation corresponds to the nontemporal subgraph operation with a basic graph pattern. To accommodate the scoring function, a vertex-map operation 2.2.9 must be applied as the second step on the results of the subgraph.

SocialScope also provides the Union, Intersection, and Difference binary operators based on the usual set semantics. The operators, as defined, assume that the two input graphs are drawn from

the same underlying social network and thus no property conflicts may arise. In addition, a special Link-Driven Minus operator provides a binary operation where the edges in the output correspond to the edges in the left operand if they don't exist in the right operand, regardless of the nodes overlap. There is no operation defined in the nontemporal algebra in Section 2.2.2 that corresponds to the Link-Driven Minus, and no temporal extension in TGA, but the same result can be achieved through a combination of edge creation and subgraph operators.

The Composition binary operator in SocialScope corresponds to the graph composition operator 2.2.12. It creates new edges, although how the edge identity is assigned is not specified. We can assume that a Skolem function is used for the purpose of creating edge ids, since they are required by the language. Consequently, the edge creation operator in TGA 3.2.14 serves as its temporal extension. A Semi-Join binary operator in SocialScope is similar to the Composition operator. It returns all edges in the left operand, and the nodes connected by those edges, such that the edges meet the directional condition in relation to edges in the right operand. There is no operator in the generalized nontemporal algebra of the TGA, corresponding to the Semi-Join, although it can be simulated with a union and subgraph operators.

Finally, SocialScope provides a Node Aggregation and Link Aggregation operators. The Node Aggregation corresponds to the nontemporal aggregation operator 2.2.8 with a basic pattern. The temporal edge creation operator 3.2.14 generalizes the Link Aggregation when it includes aggregation functions, although the SocialScope paper does not specify how edge ids are defined for the new aggregated edges.

In summary, the following TGA operators cover the temporal expression of various SocialScope nontemporal operators: union, intersection, difference, subgraph, attribute-based aggregation, vertex-map, and edge creation.

6.4.3 GraphLog

The GraphLog graph query language [31] takes graphical (drawn) graph patterns as input over labeled graphs. The output is a relation either of edges matching the distinguished edge, or nodes. A distinguished edge is an edge drawn with a bold line. If the graph pattern includes a distinguished edge, the query can be mapped to the temporal edge creation operator 3.2.14, with two differences. First, the output of the GraphLog query is a relation of new edges, rather than a graph, but we may consider the new edges to be additions to the input graph to obtain closure. Secondly,

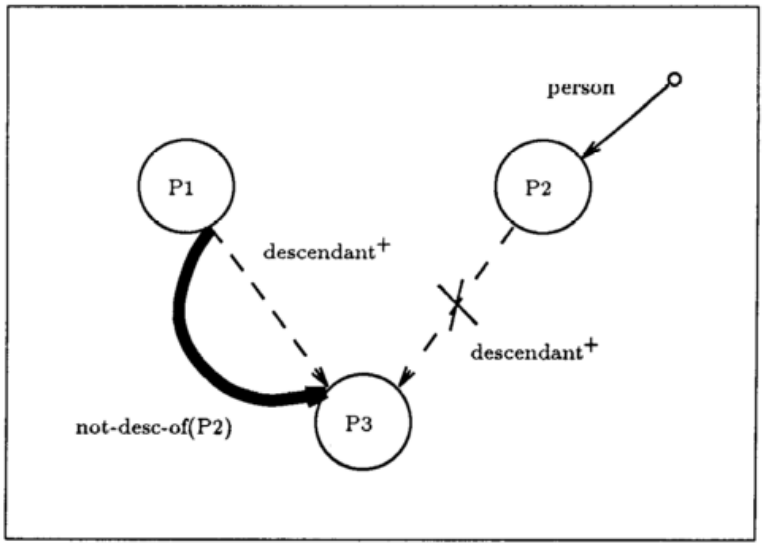


Figure 6.2: An example GraphLog query: the descendants of P1 which are not descendants of P2. [31], Figure 2, p.407

GraphLog supports negation, whereas TGA does not. Thus GraphLog queries with negation cannot be directly expressed in TGA. An example GraphLog query with negation is shown in Figure 6.2.

When there is no distinguished edge, but rather a node is drawn with a bold border, a GraphLog query corresponds to the nontemporal subgraph operator 2.2.4 with node variables assigned only to the nodes of interest in the pattern, and no edge variables.

The **G** graph query language is a pre-cursor to GraphLog that also uses graphical graph queries with regular expressions, but without a notion of a distinguished edge or negation [33]. As such, **G** queries correspond to the subgraph operation in our generalized nontemporal graph language.

6.5 Evolving Graph Systems

Three systems in the literature focus on systematic support of evolving graphs, all of them non-compositional. Miao et al. [74] developed ImmortalGraph (formerly Chronos), a proprietary in-memory execution engine for temporal graph analytics. The ImmortalGraph system has no formal model, but informally an evolving graph is defined as a series of activities on the graph, such as node additions and deletions. This is a streaming or delta approach, which is popular in temporal databases because it is unambiguous and compact. ImmortalGraph does not provide a query language, focusing primarily on efficient physical data layout. Many insights about temporal vs. structural locality by [74] hold in our setting. The batching method for snapshot analytics used by OG is similar to the one proposed in ImmortalGraph. However, despite having a distributed version,

ImmortalGraph was developed with the focus on centralized rather than distributed computation and [74] does not explore the effect of distribution on batching performance.

The G* system [62] manages graphs that correspond to periodic snapshots, with the focus on efficient data layout. It takes advantage of the similarity between successive snapshots by storing shared vertices only once and maintaining per-graph indexes. Time is not an intrinsic part of the system, as it is in TGA, and thus temporal queries with time predicates like node creation are not supported. G* provides two query languages: procedural query language PGQL, and a declarative graph query language (DGQL). PGQL provides graph operators such as retrieving vertices and their edges from disk and non-graph operators like aggregate, union, projection, and join. All operators use a streaming model, i.e., like in traditional DBMS, they pipeline. DGQL is similar to SQL and is converted into PGQL by the system.

Finally, the Historical Graph Store (HGS) system [58] is an evolving graph query system based on Spark. It uses the property graph model and supports retrieval tasks along time and entity dimensions through Java and Python API. It provides a range of operators such as selection (equivalent to our subgraph operators but with no temporal predicates), timeslice, nodecompute (similar to map but also with no temporal information), as well as various evolution-centered operators. HGS does not provide formal semantics for any of the operations it supports and the main focus is on efficient on-disk representation for retrieval.

Table 6.3 shows operator coverage in comparison to the three evolving graph systems that provide APIs. TGA is strictly more expressive than any of the three systems and is the only evolving graph algebra to provide node and edge creation as well as suport for temporal predicates.

Table 6.3: Mapping between TGA and other published systems.

Operation	**ImmortalGraph**	**G***	**HGS**
trim	Vertex and Edge iterators	VertexOperator	timeslice
subgraph	N/A	N/A	selection, non-temporal, attribute-based
map	N/A	ProjectionOperator	nodecompute, non-temporal
aggregation	indirectly through iterative computing interface	N/A	N/A
union	N/A	UnionOperator (relational, not graph)	N/A
intersection	N/A	JoinOperator (relational, not graph)	N/A
difference	N/A	N/A	N/A
attribute node creation	N/A	AggregateOperator (relational, does not create new nodes)	N/A
temporal node creation	N/A	N/A	N/A
edge creation	N/A	N/A	N/A

Chapter 7: Future Work

In this chapter, we discuss some of the limitations of our work and areas, where additional research could lead to meaningful improvements. We focus on two main points: that a sequenced semantics of operations may be desirable for TGA in place of the current point semantics (Section 7.1), and our ideas for developing a declarative language over the TGA with query optimization (Sections 7.2 and 7.3). The sequenced semantics discussion is a revision on our short paper [97].

7.1 TGA with Sequenced Semantics

7.1.1 Critique of TGraph Model

As we showed in Section 6.1, the dominant logical model for evolving graphs over the past 20 years has been a sequence of static graphs, termed *snapshots*. This model is a graph-specific adaptation of the *point-based temporal model* [88], and it introduces a semantic ambiguity that has been well studied in the temporal relational databases literature [12]: if an entity (graph, vertex or edge) with the same attributes exists in two consecutive snapshots, does it represent the same fact or two different facts? What does it mean for an entity to change?

Figure 7.1 shows an example of an evolving social network, in which vertices represent people, while edges represent interactions between them such as likes and conversations. In this example, did Alice and Bob have two conversations over the time period $[t1, t4)$ or one long one? Did Alice

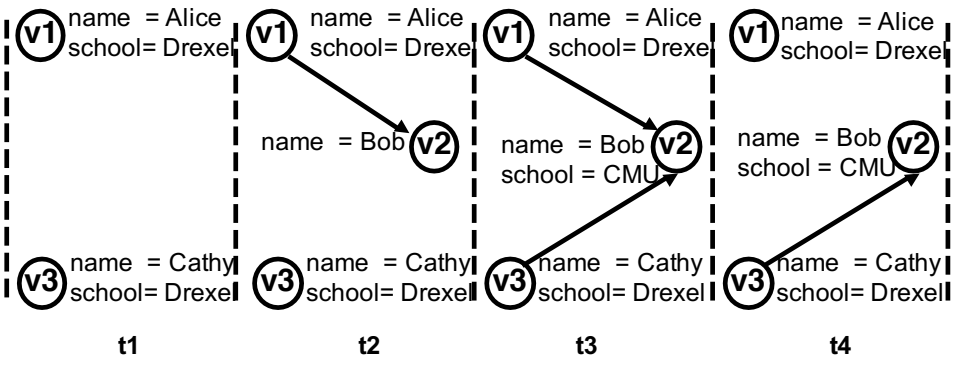


Figure 7.1: A social network as a snapshot sequence.

Table 7.1: A social network as coalesced temporal relations.

TV

v	**a**	**T**
v1	name=Alice,school=Drexel	[t1,t5)
v2	name=Bob	[t2,t3)
v2	name=Bob,school=CMU	[t3,t5)
v3	name=Cathy,school=Drexel	[t1,t5)

TE

e	**v1**	**v2**	**a**	**T**
e1	v1	v2	∅	[t2,t4)
e2	v3	v2	∅	[t3,t5)

undergo any changes during this time? Which user was the most active in this network, as defined by the number of distinct interactions? What is the rate of change of this network? We cannot answer these questions without additional information in a point-based model. Suppose that Alice held a temporary position at Drexel at time $t1$ and transferred to a permanent one at time $t2$. This information cannot be represented in the point-based model. Suppose that Alice and Bob had two short interactions, while Cathy and Bob had one longer one. The point-based model cannot distinguish between these two cases.

This kind of semantic ambiguity affects several graph operations, most notably aggregation and retrieval of change history, and, as a result, local (confined to specific entity or subset of entities) and global (whole-graph) temporal queries that are useful for evolving graph analysis.

Our TGraph model is based on the point semantics of time with interval timestamps. Point semantics is convenient for evolving graph operations because it models graph evolution over multiple properties on the level of individual graph nodes and edges. To use intervals in a time-stamped model, we coalesce, i.e., merge value-equivalent tuples over overlapping and adjacent time points [13]. Table 7.1 represents the network of Figure 7.1 as a pair of coalesced temporal relations — TV for vertices and TE for edges.

Importantly, the use of intervals to represent a sequence of value-equivalent time-adjacent snapshots is not semantically equivalent to a model with *sequenced semantics*, where entities are time-stamped with intervals that have meaning. Note that Alice shows no changes during the whole time interval, with a single tuple over $[t1, t4)$. Similarly, two interactions between Alice and Bob are coalesced into one. A work-around to avoid coalescing tuples that represent different facts is to add attributes to entities, in order to distinguish between changes and non-changes. For example, we can add position title to the vertex Alice to state that Alice changed jobs at time $t2$, and add

a conversation id to each edge to designate distinct conversations. Unfortunately, this solution is ad-hoc rather than general and does not hold up over time, as discussed in [12].

Another weakness of the point semantics of TGA is the coalescing requirement. Coalescing is an expensive operation. Lazy coalescing mitigates this problem to an extent, but, in general, the coalescing requirement has a severe impact on runtime in our prototype.

The alternative, sequenced semantics, is the de facto standard in temporal databases. It may be desirable to modify TGA to use the sequenced semantics.

7.1.2 TRA with Sequenced Semantics

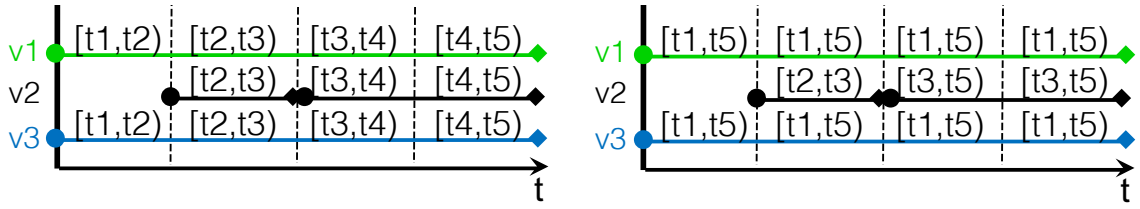
The question of semantics of temporal data has been thoroughly explored in the relational temporal database community. Böhlen et al. [12] defined point and sequenced models, and showed that the difference between the models lies in the properties of the operators, and not in the use of intervals as representational devices. With this foundation, Dignös et al. [35] defined sequenced semantics, with properties of snapshot reducibility, extended snapshot reducibility, and change preservation, and showed how non-temporal operators can be applied to temporal data. Snapshot reducibility means that a temporal operator produces the same result as an equivalent non-temporal operator over corresponding snapshots. Extended snapshot reducibility allows references to timestamps in the operators by propagating them as data. Point semantics has both of these properties as well.

The third property, change preservation, is unique to sequenced semantics. It states that operators only merge contiguous time points of a result if they have the same lineage. As shown in [35], all three properties can be guaranteed through the use of the normalize and align operators on non-temporal relations, extended with an explicit time attribute.

7.1.3 TGraph Model with Sequenced Semantics

The TGraph representation (Def. 3.1.3) does not require any modification, except the removal of Requirement R3, which requires coalescing consecutive and overlapping value-equivalent tuples in TV and TE.

The TGA algebra requires several modification to adhere to the change preservation property. For instance, the removal of the trim operator should be considered because it violates the change preservation property by directly modifying the tuple timestamps. The trim operator functionality can be partially supported through the use of vertex- and edge-subgraph with a temporal predicate.



(a) with fragments

(b) with full timestamps

Figure 7.2: Vertices from Figure 7.1 split in 4 partitions.

However, since subgraph does not modify timestamps, the overall graph history period may be different, which has a downstream effect on the temporal-window node creation operator.

Consider a growth-only graph, where new nodes and edges are added, but never removed. If we are interested in analyzing just the data from a temporal subset, subgraph will not return that subset, as some nodes and edges will have started before the input interval and all end at the same time.

Another open question is how to deal with foreign key enforcement. In the point-based TGraph model, edge timestamps may be trimmed to preserve soundness when one of the end points is removed for part of the edge period. With sequenced semantics timestamp modification of this kind is not valid. However, we cannot reject all operations that invalidate some edges, since vertex-subgraph is one such operation and is a core graph operation.

A compromise solution is to use a hybrid semantics. With the hybrid approach, most TGA operators would adhere to sequenced semantics, but the trim and window-based node creation operators would remain in point semantics. Other variants are possible.

7.1.4 Sequenced Semantics in a Distributed Environment

Many interesting static graphs are so large that they necessitate a distributed approach, as evidenced by the plethora of works on Pregel-style computation and graph partitioning [72]. In this section we discuss the challenges inherent in supporting three properties of sequenced semantics — snapshot reducibility, extended snapshot reducibility, and change preservation — in a distributed environment.

Snapshot reducibility. Evolving graphs can be partitioned among the available machines using time locality. Following convention, we refer to the operator that can produce such partitioning as a *splitter*. The splitter places each tuple (vertex or edge) into one or more partitions based on its timestamp. The goal of the splitter is to form partitions that are balanced, i.e., have approximately the same number of items, under the assumption that most operations can be executed locally at

each partition. Recall that snapshot reducibility requires a temporal operator to produce the same result as if it were evaluated over each snapshot. Validity period of a tuple that spans more than one temporal partition is split, and the tuple is replicated across partitions. This increases the overall size of the relation, but all operations can now be carried out within each partition. See Figure 7.2a for a simple example of the V relation being split into four temporal partitions.

For the purposes of illustration, consider the vertex-subgraph operation (Def. 3.2.7). Observe that we can carry out the subgraph operation with non-temporal predicates, e.g., name='Alice', at each partition individually, without any cross-partition communication.

The question of optimal splitting has been addressed by Le et al. [64], who demonstrated that a temporal relation can be efficiently split into k buckets in cases of both internal memory and external memory, and guarantee optimality of the solution. This method requires a sequential scan of the relation to compute an index called the stabbing count array. How to make this method more efficient in a distributed environment is an open question.

The subgraph operation requires co-partitioning of graph relations to enforce referential integrity on edges. A number of alternatives for co-partitioning present themselves, as the vertex, edge and attribute relations are not guaranteed to have the same splitters due to different evolution rates. Typically, vertices are co-partitioned with edges in the non-temporal case [44], and this likely is most efficient with evolving graphs as well. This is how snapshot groups are computed in Portal.

Extended snapshot reducibility. Snapshot reducibility can be guaranteed in the distributed setting, as shown above for a subgraph query without temporal predicates. In general, a subgraph query may explicitly reference temporal information in compliance with the extended snapshot reducibility property of sequenced semantics. Refer back to Figure 7.2a and assume time granularity of years. If we perform the subgraph operation, selecting vertices that persist for longer than 2 years, over the split then we will get no matches. However, the original relation contains two matches – only Bob does not meet the predicate. To support extended snapshot reducibility over a split relation, during partitioning tuples should be placed into their partitions with their full original timestamps. Incidentally, this is what Le at al. describe in their work on optimal splitters [64]. Figure 7.2b shows relation V split in the same four partitions with this approach. The above argument holds for nonrecursive subgraph only. If a recursive pattern is used, the computation cannot be localized to a single partition.

Change preservation. Change preservation property requires that derived tuples should only be coalesced if they share lineage. To support this property, normalize and align operators are

used [35]. The normalize operator splits each tuple in the input relation with respect to a group of tuples such that each timestamp fragment is either fully contained or disjoint with every timestamp in the group. The align operator splits each tuple with respect to a group of tuples such that each timestamp fragment is either an intersection with one of the tuples in the group or is not covered by any tuple in a group.

The normalize operator splits each tuple with respect to a group defined by the operation. For example, consider the attribute-based node creation operation (Def. 3.2.9). This operation allows the user to, for example, generate a TGraph in which vertices correspond to disjoint groups of vertices in the input that agree on the values of all grouping attributes. For instance, $\mathsf{node}_a^T(\mathsf{school}, \mathcal{G})$ will compute a vertex for each value of TV.a.school. While the group defined for each tuple (distinct value of school) spans temporal partitions, only tuples within the same partition overlap. Unfortunately, full information about the group is required at each partition because of the untrimmed timestamps. Thus, the normalize and align operations cannot be carried out locally at each partition without additional information.

An important challenge to address is how to efficiently support node creation over temporal windows in a distributed setting. This operation requires cross-partition communication, which impacts the cost model, requiring a generalization of the approach of Le et al. [64]. Similarly, any recursive operation, e.g. subgraph, node creation, edge creation, aggregation, also spans multiple partitions and requires cross-partition communication.

7.2 Declarative Language

This dissertation proposed an algebra, which the Portal system supports through a Scala API. We begun work on a declarative language for TGA. However, the mix of the temporal and graph aspects in a closed algebra make such an endeavor far from straight-forward. Many graph query languages themselves do not return graphs from queries and cannot be used as models for this work. For example, Cypher, a query language of the Neo4j database [78], returns essentially a relation. Cypher does allow the user an ability to specify complex navigational patterns, which is also required in TGA. For instance, vertex-subgraph accepts a temporal navigation graph pattern. It remains an open question how to represent such a pattern and embed it in the query.

To provide a starting place for a declarative language, we show queries for the three of our motivating use cases (Section 1.3).

Example 41. *To start, load a 5-year subset of the data:*

```
Create TGraph T1 As {
   VSelect * ESelect *
   From 'path/to/data' as wiki
   When start>='2010' AND end<'2015'}
```

The corresponding algebraic expression in TGA uses the trim *operator (Def. 3.2.3):* $\mathcal{G}_1 = \mathit{trim}^T_{[2010,2015)}(wiki)$.

Example 42. *Compute a temporally aggregated view of T1 into 6-month windows. A window includes nodes and edges that correspond to users who communicated regularly: a node and an edge are each present if they exist in every snapshot during the 6-month period.*

```
Create TGView T2 As {
   VSelect * ESelect *
   From T1
   TGroup By 6 months
   VExists always EExists always }
```

*This query corresponds to window-based node creation (Def. 3.2.10). Users can specify node and edge quantifiers (*always, most, at least n, *and* exists*) on the minimum lifetime of the entity within the window in order for it to be included in the result.*
$\mathcal{G}_2 = \mathit{node}^T_w(\mathsf{w} = 6\ \mathsf{mon}, \mathsf{r_v} = \mathsf{always}, \mathsf{r_e} = \mathsf{always}, \mathcal{G}_1)$.

Example 43. *Compute connected components at each time point.*

```
Create TGView T3 As {
   VSelect components() as comp, *  ESelect *
   From T2 }
```

This query correspond to the following TGA expression: $\mathcal{G}_3 = \mathsf{agg}^T(P, \mathcal{G}_2)$*, where* P *is a pattern we showed previously (Figure 3.9). Note, however, that we are treating the connected components analytic as a special buit-in function here.*

Note the use of $*$ after invocation of the components analytic. With the property model, different nodes may have different properties, so listing them all may be burdensome for the user.

Example 44. *Generate a new* TGraph*, in which a node corresponds to a connected component, and compute the size of the connected component.*

```
Create TGView T4 As {
   VSelect count(*) as cnt  ESelect *
   From T3
   VGroup By comp }
```

This operation allows the user to generate a TGraph in which a vertex corresponds to a group of vertices in the input that agree on the values of all grouping attributes, using the attribute-based node creation operation (Def. 3.2.9). The corresponding TGA expression is $\mathcal{G}_4 = \mathsf{node}_a^T(\mathsf{g} = \mathsf{comp}, \mathsf{f_v} = \mathsf{count}(1), \mathcal{G}_3)$.

Example 45. *Filter out vertices that represent communities too small to be of interest (e.g., of 1-2 people).*

```
Create TGView T5 As {
   VSelect * ESelect *
   From T4
   VWhere cnt > 2 }
```

This query corresponds to a TGA expression that invokes the subgraph operation $\mathcal{G}_5 = \mathit{subgraph}^T(\mathsf{v.a.count} > 2, \mathcal{G}_4)$.

The queries of Examples 41 — 45 can be combined into a single complex query:

```
VSelect * ESelect *
From (VSelect count(*) as cnt  ESelect *
      From (VSelect components() as comp, * ESelect *
            From 'path/to/data' as wiki
            When start>='2010' AND end<'2015'
            TGroup By 6 months
            VExists always EExists always)
      VGroup By comp)
VWhere cnt > 2
```

In a Portal query, trim and subgraphs take precedence over the other operations, followed by node creation, and then by aggregation.

Another use case demonstrates some of the remaining operations of TGA. DBLP and arXiv datasets contain co-authorship information and can provide interesting insights into the dynamics of the CS research community. Nodes represent authors, and edges — a joint publication between a pair of authors.

Example 46. *Assuming that author ids are drawn from the same domain, i.e., are assigned consistently in DBLP and arXiv, we can combine the two datasets for a more comprehensive view using temporal graph union (Def. 3.2.11):*

```
Create TGView T6 As {
   VSelect left(name)  ESelect max(cnt)
   From dblp Union arXiv }
```

The corresponding TGA expression is

$\mathcal{G}_6 = dblp \ \cup^T_{f_v=left(name),f_E=max(cnt)} \ arXiv.$

Unlike in relational algebra, TGA union requires aggregate functions to resolve multiple property values for the same node or edge. If a vertex exists in both DBLP and arXiv, left takes the value of the vertex property name from the left operand, DBLP.

The syntax of the above examples is SQL-inspired and is a reasonable starting point. On the one hand, this should make it easier to learn for people familiar with regular databases. On the other, these examples sidestep the question of pattern specification. To get the full functionality of TGA, such specification is necessary. We believe that languages such as Cypher and SPARQL should be more closely examined for this purpose.

7.3 Query Optimization

As we showed in our experiments (Section 5.4), no single in-memory representation is best suited to all TGA operations, although the RG representation can be safely discarded. This opens the door to cost-based query optimization in order to select the representation most likely to lead to good performance on the overall query, or to switch between representations when it makes sense to do so. Other kinds of possible optimizations include:

- Attribute pruning (column pruning in SQL) when the query does not refer to all graph properties.
- Collapse of multiple filters into one.
- trim can be pushed to the bottom of the tree, except when it appears after temporal window node creation.
- trim can be added to each subtree of the query execution plan of temporal intersection, based on the information about temporal ranges of TGraphs stored in the system catalog.

- Unlike selection in SQL, filters cannot be pushed down through union because of the aggregation functions.

Future work is needed to define all TGA operator equivalencies and reordering rules, as well as to generate a cost model based on the data characteristics, such as the graph evolution rate.

Chapter 8: Conclusions

The goal of our research has been to provide a systematic support for querying and exploratory evolving graph analysis. Previous efforts pursued ad hoc approaches to modeling evolving graphs by representing time as data or using a snapshot sequence model, with all inherent limitations. As a result, while the types of analyses on evolving graphs are numerous in the literature, there is no unifying model or a query language. Previous efforts to provide *systematic* support have addressed two main areas: efficient snapshot retrieval [57, 58] and efficient snapshot analytics [74, 96]. This still leaves many other kinds of analyses, from discovery of spatio-temporal patterns, to changing the temporal resolution of the data for a different look, to combining multiple evolving graph sources into one.

This dissertation proposed a model for evolving graphs with a concise set of operations over a logical model based on the temporal graph algebra. We combined insights from the temporal relational algebra and from graph algebras, and used a principled approach to cover a large set of published analysis types in a general way. The model we propose has the desired theoretical properties of snapshot reducibility, extended snapshot reducibility, and, with an appropriate syntax, can be made temporally semi-complete. We show that the temporal graph algebra we propose is strictly more expressive than the three published systems, G* [62], ImmortalGraph [74], and HGS [58]. We also provide an initial formal analysis of its expressive power. Further analysis is beneficial and can build on the foundation laid in this dissertation.

We created a platform, Portal, that data scientists and researchers can use on large evolving graph datasets in a distributed environment. We are releasing Portal to the public as an open-source project, and are the first to do so in the evolving graph community. Because Portal is developed within the popular Apache Spark architecture and provides an easy integration with SparkSQL, a rich set of queries will be within reach of a wide audience.

One of the main insights of the system work in this dissertation is building on the ideas of temporal locality, but showing that a hybrid layout provides better performance on several important queries, e.g., snapshot analytics. We experimentally show that a naive implementation based on a

sequence of snapshots not only cannot support all operations, but also scales poorly as the evolving graph timeline is extended. The lazy coalescing and foreign key enforcement rules we developed can be used in any environment and are not limited to our architecture.

Our experiments show that Portal provides a more efficient approach to analytics than several baselines: a naive implementation, the G* system, and a simulated ImmortalGraph system. We support operations on billion-edge graphs in minutes, on a cluster of modest size. We also show how interesting use cases can be expressed in the temporal graph algebra and executed in Portal to yield insights about a common interaction network.

It is our hope that this work will be the next step in making evolving graph analysis widely accessible to researchers from different domains and speed their discovery process.

Bibliography

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

[2] Neo4j: What is a graph database? `https://neo4j.com/developer/graph-database/#property-graph`. [Online; accessed 7-July-2016].

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[5] James F Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[6] Sihem Amer-Yahia, Laks Lakshmanan, and Cong Yu. SocialScope: Enabling information discovery on social content sites. In *CIDR*, 2009.

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern graph query languages. *CoRR*, abs/1610.06264, 2016.

[8] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008.

[9] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL : Relational Data Processing in Spark. In *SIGMOD'15*, Melbourne, Australia, 2015.

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

[11] Antje Beyer, Peter Thomason, Xinzhong Li, James Scott, and Jasmin Fisher. Mechanistic insights into metabolic disturbance during type-2 diabetes and obesity using qualitative networks. *T. Comp. Sys. Biology*, 12:146–162, 2010.

[12] Michael H Böhlen, Renato Busatto, and Christian S Jensen. Point Versus Interval-based Temporal Data Models. In *Proceedings of the Fourteenth IEEE International Conference on Data Engineering*, pages 192–200, Orlando, Florida, 1998.

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

[14] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. How would you like to aggregate your temporal data? In *TIME*, 2006.

[15] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. Evaluating the Completeness of TSQL2. In *Recent Advances in Temporal Databases, International Workshop on Temporal Databases*, pages 153–172, 1995.

[16] Michael H Böhlen, Christian S Jensen, and Richard T Snodgrass. Temporal Statement Modifiers. *ACM Transactions on Database Systems*, 25(4):407–456, 2000.

[17] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. *Temporal Compatibility*, pages 2936–2939. Springer US, Boston, MA, 2009.

[18] Karsten M. Borgwardt, Hans Peter Kriegel, and Peter Wackersreuther. Pattern mining in frequent dynamic subgraphs. In *Proceedings - IEEE International Conference on Data Mining, ICDM*, pages 818–822, Hong Kong, 2006.

[19] Oswald Campesato and Kevin Nilson. *Web 2. 0 Fundamentals: with AJAX, Development Tools, and Mobile Platforms*. Jones & Bartlett Learning, 2010.

[20] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-Varying Graphs and Dynamic Networks. In *Proceedings of the 10th international conference on Ad-hoc, mobile, and wireless networks*, volume 6811, pages 346–359, 2011.

[21] Ciro Cattuto, Marco Quaggiotto, André Panisson, and Alex Averbuch. Time-varying social networks in a graph database. In *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13*, volume 1, pages 1–6, 2013.

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

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

[24] Sudarshan S Chawathe, Serge Abiteboul, and Jennifer Widom. Representing and Querying Changes in Semistructured Data. In *ICDE*, Orlando, Florida, USA, 1998.

[25] Sudarshan S Chawathe, Serge Abiteboul, and Jennifer Widom. Managing Historical Semistructured Data. *Theory and practice of object systems*, 24(4):1–20, 1999.

[26] Junghoo Cho and H Garcia-Molina. The evolution of the web and implications for an incremental crawler. *VLDB '00 Proceedings of the 26th International Conference on Very Large Data Bases*, pages 200–209, 2000.

[27] Jan Chomicki. Temporal query languages: A survey. In *Temporal Logic, First International Conference, ICTL '94, Bonn, Germany, July 11-14, 1994, Proceedings*, pages 506–534, 1994.

[28] Jan Chomicki and David Toman. Temporal relational calculus. In *Encyclopedia of Database Systems*, pages 3015–3016. Springer UxS, Boston, MA, 2009.

[29] James Clifford, Albert Croker, and Alexander Tuzhilin. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems*, 19(1):64–116, 1994.

[30] James Clifford and Abdullah Uz Tansel. On an Algebra for Historical Relational Databases: Two Views. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 247–265, 1985.

[31] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 404–416, New York, NY, USA, 1990. ACM.

[32] Mariano P. Consens and Alberto O. Mendelzon. Low-complexity aggregation in graphlog and datalog. *Theoretical Computer Science*, 116(1):95 – 116, 1993.

[33] Isabel F Cruz, Alberto Mendelzon, and Peter T. Wood. A GRAPHICAL QUERY LANGUAGE SUPPORTING RECURSION +. In *SIGMOD*, pages 323–330, 1987.

[34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplied Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[35] Anton Dignös, Michael H Böhlen, and Johann Gamper. Temporal Alignment. In *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*, pages 433–444, Scottsdale, Arizona, USA, 2012.

[36] Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John Miller. Towards Efficient Query Processing on Massive Time-Evolving Graphs. In *Proceedings of the 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 567–574, 2012.

[37] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Rec.*, 26(3):4–11, September 1997.

[38] Afonso Ferreira. Building a reference combinatorial model for MANETs. *IEEE Network*, 18(5):24–29, 2004.

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

[40] Betsy George, James M Kang, and Shashi Shekhar. Spatio-Temporal Sensor Graphs (STSG): A Data Model for the Discovery of Spatio-Temporal Patterns. *Intelligent Data Analysis*, 13(3):457–475, 2009.

[41] Betsy George and Shashi Shekhar. Time-aggregated graphs for modeling spatio-temporal networks. *Journal on Data Semantics*, 11:191–212, 2006.

[42] Michaela Goetz, Jure Leskovec, Mary McGlohon, and Christos Faloutsos. Modeling blog dynamics. In *ICWSM*, 2009.

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

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

[45] Wentao Han, Wenguang Chen, and Enhong Chen. Chronos : A Graph Engine for Temporal Graph Analysis. In *EuroSys*, Amsterdam, Netherlands, 2014.

[46] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

[47] Frank Harary. *Graph theory.* Addison Wesley Publishing Co, 1969.

[48] Jonathan Hayes and Claudio Gutierrez. *Bipartite Graphs as Intermediate Model for RDF*, pages 47–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[49] Huahai He and Ambuj K. Singh. Graphs-at-a-time : Query Language and Access Methods for Graph Databases Categories and Subject Descriptors. In *Proceedings of the 28th International Conference on Management of Data, SIGMOD*, pages 405–417, 2008.

[50] Ping-yu Hsu and D Stott Parker. Improving SQL with Generalized Quantifiers. In *ICDE*, 1995.

[51] Wenyu Huo and Vassilis J Tsotras. Efficient Temporal Shortest Path Queries on Evolving Social Graphs. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14, pages 38:1–38:4, New York, NY, USA, 2014. ACM.

[52] Christian S. Jensen and Richard T. Snodgrass. Snapshot equivalence. In *Encyclopedia of Database Systems*, pages 2659–2659. Springer US, Boston, MA, 2009.

[53] Christian S. Jensen and Richard T. Snodgrass. Temporal data models. In *Encyclopedia of Database Systems*, pages 2952–2957. Springer US, Boston, MA, 2009.

[54] Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass. Unifying temporal data models via a conceptual model. *Information Systems*, 19(7):513 – 547, 1994.

[55] Andrey Kan, Je Chan, James Bailey, and Christopher Leckie. A Query Based Approach for Mining Evolving Graphs. In *Eighth Australasian Data Mining Conference (AusDM 2009)*, volume 101, Melbourne, Australia, 2009.

[56] Arush Kharbanda. Apache spark: A look under the hood. `https://www.sigmoid.com/apache-spark-internals/`. [Online; accessed 27-April-2017].

[57] Udayan Khurana and Amol Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 997 – 1008, Brisbane, QLD, 2013.

[58] Udayan Khurana and Amol Deshpande. Storing and Analyzing Historical Graph Data at Scale. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT'16*, pages 65–76, Bordeaux, France, 2016.

[59] Graham Klyne, Jeremy J. Carroll, and Brian McBride. Rdf 1.1 concepts and abstract syntax. w3c recommendation. Standard, W3C, February 2014.

[60] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura. On Graph Deltas for Historical Queries. In *Proceedings of 1st Workshop on Online Social Systems (WOSS) 2012, in conjunction with VLDB 2012*, Istanbul, Turkey, 2012.

[61] Krishna G. Kulkarni and Jan-Eike Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[62] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Sean R. Spillane, Jayadevan Vijayan, Jeong Hyon Hwang, and Wook Shin Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 33(4):479–514, 2014.

[63] M. Lahiri and T.Y. Berger-Wolf. Mining Periodic Behavior in Dynamic Social Networks. In *2008 Eighth IEEE International Conference on Data Mining*, pages 373–382, 2008.

[64] Wangchao Le et al. Optimal splitters for temporal and multi-version databases. In *SIGMOD*, 2013.

[65] Kristina Lerman, Marina Rey, Rumi Ghosh, and Jeon Hyung Kang. Centrality Metric for Dynamic Networks. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 70–77, Washington, D.C., 2010.

[66] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The dynamics of viral marketing. *TWEB*, 1(1), 2007.

[67] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. Microscopic evolution of social networks. In *ACM SIGKDD*, pages 462–470, 2008.

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

[69] Panagiotis Lionakis, Kostas Stefanidis, and Georgia Koloniari. SQTime : Time-enhanced Social Search Querying. In *33rd International Conference on Conceptual Modeling*, Atlanta, GA, 2014.

[70] Ling Liu and M. Tamer Zsu. *Encyclopedia of Database Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[71] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, pages 135–146, 2010.

[72] Robert Ryan McCune et al. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM CSUR*, 1(1), 2015.

[73] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *36th International Conference on Very Large Data Bases*, pages 330–339, 2010.

[74] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Transactions on Storage*, 11(3):14–34, 2015.

[75] Angelo Montanari and Jan Chomicki. *Time Domain*, pages 3103–3107. Springer US, Boston, MA, 2009.

[76] Panagiotis Papadimitriou, Ali Dasdan, and Hector Garcia-Molina. Web graph similarity for anomaly detection. *J. Internet Services and Applications*, 1(1):19–30, 2010.

[77] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On Querying Historical Evolving Graph Sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[78] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. O'Reilly Media, Inc., 2013.

[79] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, jun 1999.

[80] Purnamrita Sarkar, Deepayan Chakrabarti, and Michael I. Jordan. Nonparametric link prediction in dynamic networks. In *ICML*, 2012.

[81] Konstantinos Semertzidis, Kostas Lillis, and Evaggelia Pitoura. TimeReach: Historical Reachability Queries on Evolving Graphs. In *Proceedings of the 18th International Conference on Extending Database Technology*, pages 121–132, Brussels, Belgium, 2015.

[82] Alexander Shkapsky, Mohan Yang, Matteo Iterlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD'16*, pages 1135–1149, San Francisco, CA, 2016.

[83] Richard Snodgrass and Ilsoo Ahn. A taxonomy of time databases. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, pages 236–246, New York, NY, USA, 1985. ACM.

[84] Kumar Sricharan and Kamalika Das. Localizing anomalous changes in time-evolving graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1347–1358, Snowbird, Utah USA, 2014.

[85] Joshua M. Stuart, Eran Segal, Daphne Koller, and Stuart K. Kim. A gene-coexpression network for global discovery of conserved genetic modules. *Science*, 5643(302):249—255, 2003.

[86] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1887–1901, Melbourne, Victoria, Australia, 2015.

[87] Björn Titz, Seesandra V. Rajagopala, Johannes Goll, Roman Häuser, Matthew T. McKevitt, Timothy Palzkill, and Peter Uetz. The binary protein interactome of treponema pallidum – the syphilis spirochete. *PLOS ONE*, 3(5):1–11, 05 2008.

[88] David Toman. Point-stamped temporal models. In *Encyclopedia of Database Systems*, pages 2119–2123. Springer US, Boston, MA, 2009.

[89] Leslie GLeslie 6. Valiant 103 Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[90] André Vermeij. Apple's internal innovation network unraveled – part 1 – evolving networks. `https://www.kenedict.com/apples-internal-innovation-network-unraveled-part-1-evolving-networks/`. [Online; accessed 16-December-2015].

[91] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[92] Peter T. Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, mar 2012.

[93] Zhe Wu. Bridging rdf graph and property graph data models. *8th LDBC Technical User Community Meeting*, 2016.

[94] Lei Yang, Lei Qi, Yan-ping Zhao, Bin Gao, and Tie-yan Liu. Link Analysis using Time Series of Web Graphs. In *Proceedings of Conference on Information and Knowledge Management (CIKM)*, pages 1011–1014, Lisboa, Pprtugal, 2007.

[95] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets : A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

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

[97] V. Zaychik Moffitt and J. Stoyanovich. Towards sequenced semantics for evolving graphs. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT'17*, 2017.

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