

Eingereicht von
Matthias Steinbauer

Angefertigt am
Institut für
Telekooperation

Erstbeurteiler
Prof. Gabriele
Anderst-Kotsis

Zweitbeurteiler
Prof. Paul Spirakis

December 2016

DynamoGraph: Large-scale Temporal Graph Processing and its Application Scenarios



Dissertation
zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften
im Doktoratsstudium der
Technischen Wissenschaften

To Lena and Sonja
your love is a constant source of inspiration!
I am truly thankful to have you in my life!

Affidavit / Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, December 2016

Matthias Steinbauer

Acknowledgment

First of all I would like to thank Sonja, the love of my life, mother of our child Lena. Only due to the continuous support, the persistent interest in my research progress and the loving environment you created at home, this work has been possible. This work is for you and all the joy you have brought to my life!

I would like to specifically thank my thesis advisor Prof. Gabriele Anderst-Kotsis who has already influenced me during my early days as a computer science student. She has discovered my interest in research. She is an outstanding role model for a career as a computer scientist and has always been a great source of support and feedback to me. I do not know a single person in my life who can give better, on the spot feedback but her!

Also, I would like to thank Prof. Paul Spirakis my second thesis advisor. He has provided fundamental research in the area of my thesis and thus is one of the giants on whose shoulders I am standing. Thank you for the excellent feedback and the great interest in my work.

The ones who really set the stepping stone for my career in computer science are my parents Roswitha and Siegfried. It has been them that discovered my interest in computing machinery early on, nurtured it and finally supported my decisions to take the necessary career steps towards becoming a computer scientist. They have taught me through their example, that hard work in a field you love will lead to great achievements.

Finally, I would like to thank colleagues and friends I have made at the department: Angela, Angelika, Andrea, Andreas, Elisabeth, Herbert, Ismail, Jürgen, Karin, Martin, Paul, Sabine, Stefan, Sylva, Werner, and Wieland. Thank you for listening to my thoughts and helping me put my research into perspective. You have been a great source of motivation!

Abstract

When computers (people) are networked, their power multiplies geometrically. Not only can people share all that information inside their machines, but they can reach out and instantly tap the power of other machines (people), essentially making the entire network their computer. – Scott McNeely

Scott McNeely’s vision became reality, we have now implemented a networked world. Machines are networked, humans form networks and we are at the brink of all and everything in our lives being connected in the Internet of things. While network structures (graphs) are very well studied, in this thesis it is argued that for many problems not only structure is of interest but the change in this structure might be of even higher relevance e.g. the events of individuals joining or leaving groups in a social network. It is discussed that, although the formal concepts of changing graphs (dynamic graph) and graphs which track all changes made to them over time (temporal graphs) exist for a long time, up to now comparatively little use of these concepts is made in applied computing.

One of the reasons for this is that the application areas often are of very large size such that actually large-scale temporal graphs are the data structure of concern. For large datasets horizontally scaled computing systems based on cloud computing infrastructure, are the prevalent approach to gain speedup and to solve problems in feasible time.

This thesis discusses a distributed computing framework called *DynamoGraph* which allows to partition large-scale temporal graph data over multiple compute nodes. Vertices are at the core of the used data model and are implemented using temporal maps, a map data-structure that tracks temporal meta-information of the stored data. This mechanism is exploited in an extension of the Pregel graph processing paradigm which makes it applicable for temporal graphs. The work shows the practical feasibility with case-studies and the general scalability (compute resources and data size) of the approach. The experimental performance evaluation shows that the distributed computing approach provides significant gains and thus is inevitable in real-world settings.

Zusammenfassung

When computers (people) are networked, their power multiplies geometrically. Not only can people share all that information inside their machines, but they can reach out and instantly tap the power of other machines (people), essentially making the entire network their computer. – Scott McNeely

Scott McNeely's Vision ist Realität geworden, wir befinden uns in der Era einer vernetzten Welt, in der Menschen, Computer und auch ganz einfach Dinge im Internet of Things miteinander vernetzt sind. Obwohl Netzwerk-Datenstrukturen (Graphen) ein sehr genau untersuchtes Forschungsfeld darstellen, stellt sich heraus, dass in vielen Anwendungen gerade die Änderung in der Netzwerkstruktur von besonderer Bedeutung ist z.B. Personen die einer Gruppe beitreten. Obwohl die formalen Konzepte für veränderliche Graphen (dynamischer Graph) und auch für Graphen die diese zeitliche Änderung aufzeichnen (temporaler Graph) schon lange existieren gibt es bisher vergleichsweise wenige Studien in der angewandten Informatik.

Ein Grund dafür ist sicherlich, dass die angesprochenen Anwendungsszenarien nicht nur nach temporalen Graphen verlangen, sondern üblicherweise auch aus sehr großen Datensätzen bestehen, so dass sehr große temporale Graphen verarbeitet werden müssen. Um Datenverarbeitung auf immer weiter wachsende Datenmengen skalieren zu können ist das sogenannte horizontale Skalieren das aktuell bevorzugte Modell. Mehrere Rechner, meist auf Basis von Cloud Infrastruktur, werden im Verbund verwendet.

In dieser Arbeit wird *DynamoGraph*, ein Softwaresystem für verteiltes Rechnen auf großen temporalen Graphen, als Lösungsansatz vorgestellt. Mit Hilfe von *DynamoGraph* ist es möglich temporale Graphdaten auf Basis einer temporalen Indexdatenstruktur in einem Rechnerverbund zu verteilen. Die Datenstruktur funktioniert analog zu existierend Indexdatenstrukturen speichert aber Metadaten zur Dimension Zeit, dies kann später wiederum in der, auch in dieser Arbeit diskutierten Pregel Implementierung für temporale Graphen, verwendet werden zum Beispiel um Daten aus bestimmten Zeitfenstern zu verarbeiten.

Der Einsatz von *DynamoGraph* in praktischen Fallbeispielen zeigt, dass der Ansatz von Softwareentwicklern in der Praxis, mit vergleichsweise flacher Lernkurve, eingesetzt werden kann. Weiters wird gezeigt, dass die Methode aus dem verteilten Rechnen, Skalierbarkeit der Rechner-Ressourcen aber auch des Datensatzes erlaubt. Performance Tests zeigen, dass die Zugewinne durch das verteilte Rechnen so hoch sind, dass es in der praktischen Anwendungen unumgänglich ist.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Open Problems in Large Temporal Graph Processing	3
1.3	Preliminaries	4
1.3.1	Graphs and Temporal Graphs	4
1.3.2	Static Metrics and Analysis	7
1.3.2.1	Centrality	7
1.3.2.2	Paths and Reachability	9
1.3.3	Important Subgraphs	10
1.3.3.1	Connected Components	10
1.3.3.2	Clusters and Communities	11
1.3.3.3	Graph Partitioning	14
1.3.4	Cloud Computing and Big Data	15
1.4	Addressed Application Areas	16
1.4.1	Social Network Analysis	16
1.4.1.1	Sizes of Social Networks	17
1.4.1.2	Communication Networks and Group Performance	18
1.4.1.3	Co-Author and Co-Actor Networks	18
1.4.1.4	Criminal Networks	19
1.4.1.5	Learning Networks	20
1.4.1.6	Political Networks	20
1.4.2	Computer Networks and the Web	21
1.4.3	Cartography and Maps	22
1.5	Requirements	23
1.5.1	Temporal Dimension	23
1.5.2	Scalability	23
1.5.3	Fault Tolerance	24
1.6	Hypothesis	24
1.6.1	Temporal Graph Partitioning	24
1.6.2	Distributed Temporal Graph Storage	25
1.6.3	Distributed Temporal Graph Processing	25
1.7	Contribution	26

1.8	List of Publications	27
1.9	Outline	29
2	Related Work	30
2.1	Temporal Graphs	30
2.1.1	Temporal Graph Metrics	31
2.1.2	Dynamic Graph Algorithms	32
2.1.3	Temporal Graphs and Graph Databases	34
2.2	Large Graphs	36
2.3	Parallel and Distributed Computing	38
2.3.1	Distributed Matrix Processing	38
2.3.2	High-Performance Computing Model	39
2.3.3	Cloud Computing and Big Data	41
2.4	<i>DynamoGraph</i> Distinctive Features	43
3	Distributed Temporal Graph Processing Framework	45
3.1	Temporal Maps as Data Structure	45
3.2	Graph Partitioning Strategies	49
3.3	Parallel Computing Models	53
3.3.1	Shared-Memory Model	53
3.3.2	Parallel Message-Passing Model	54
3.3.3	Dynamic Message-Passing Model	55
3.3.4	Compute Aggregate Broadcast Computing	56
3.4	Distributed Temporal Graph Processing	57
3.4.1	Pregel-style Job Execution	59
3.4.2	Extensions over Pregel	60
3.4.3	Vertex Local Computation	62
3.4.4	Synchronisation and Halting	63
3.4.5	Algorithm Initialisation and Global Computation	64
3.4.6	Multiple Phase Algorithms	65
3.5	Triggered Algorithm Execution	66
3.6	Fault Tolerance and Reorganisation	66
3.7	Selected Example Algorithms and their Processing	67
3.7.1	Max- Min- Vertex Degree	68
3.7.2	Shortest Path	69
3.7.3	Page Rank	72
3.7.4	Label Propagation Community Detection	74
3.8	Query Methods Implemented as Supersteps	77
3.8.1	Single Vertex Retrieval	78
3.8.2	Queries for Vertex Attributes	78
3.8.3	Network Query Algorithms	79
3.9	Summary	79

4	Reference Implementation of the Distributed Processing Framework	81
4.1	Extended Requirements	82
4.1.1	Configurability	82
4.1.2	Scalability	82
4.1.3	Fault Tolerance	83
4.1.4	Multi-Tenancy	83
4.1.5	Monitoring	84
4.1.6	Modularity	84
4.1.7	Persistence	85
4.1.8	Dynamic Code Loading	85
4.2	Architectural Overview	85
4.2.1	Big Data and Cloud-based Computation	87
4.2.2	Technological Decisions	89
4.3	Distributed Configuration with ZooKeeper	89
4.3.1	Master Role Compute Nodes and their Election	91
4.3.2	Worker Role Compute Nodes	92
4.3.3	Node Failure and Recovery	94
4.4	Intra-Cluster Communication	95
4.5	Algorithm Execution	96
4.6	Client API	98
4.7	Java Dynamic Code Loading	101
4.8	Monitoring	104
4.9	Persistence Backends	106
4.9.1	File Based Persistence Backend	107
4.9.2	Cassandra Persistence Backend	108
4.10	Future Improvements	108
4.11	Concluding Remarks	109
5	Usage of the Distributed Processing Framework	110
5.1	Typical Use-Cases	110
5.2	Execution Modes	111
5.2.1	Cluster Mode	112
5.2.2	Local Developer Mode	112
5.3	Web-based Graph Visualization	113
5.4	Setup and Configuration	120
5.4.1	Apache ZooKeeper	121
5.4.2	DynamoGraph Master and Workers	121
5.4.3	Optional Persistence Backends	123
5.4.4	Optional Web Interface	123
5.5	Data-set Import	124
5.6	Case Studies	128
5.6.1	Global Social Learning Network Analytics	128

5.6.2	Temporal Political Network Analysis	130
5.7	Prototype Summary	133
6	Results and Discussion	134
6.1	Evaluation Objectives	134
6.2	Evaluation Methods	135
6.2.1	Real-World Datasets	135
6.2.2	Artificial Graph Models	136
6.3	Testbeds	139
6.3.1	OpenStack Private Cloud	140
6.3.2	Amazon AWS Public Cloud	141
6.3.3	Application Roll-out	142
6.4	Results	144
6.4.1	PageRank over Enron e-Mail Database	146
6.4.2	Multiple Algorithms on IU Click Data	149
6.4.3	Social Networks vs. Click Graphs	154
6.5	Discussion	156
7	Conclusions and Future Work	158
7.1	Future Work	158
7.1.1	Visualisation and Tooling	158
7.1.2	Performance Improvements	159
7.1.3	Dynamic Notion of Time	160
7.1.4	Incremental Computation	161
7.2	Concluding Remarks	161
A	Tooling	163
A.1	Java, JavaScript, Web	163
A.1.1	Distributed Java Applications	165
A.1.2	Client API Methods	166
A.1.3	Web-based User Interface	171
A.2	Cloud Stacks	172
B	Used Datasets	173
B.1	MIT Reality Commons	173
B.2	Enron e-Mail database	175
B.3	Click Dataset	176
	Bibliography	178

Abbreviations

- BSP** Bulk Synchronous Parallel (A computing model for distributed parallel computing with strict barrier synchronisation)
- CAB** Compute Aggregate Broadcast (A computing model in parallel computing where computation is strictly partitioned in the three phases compute, aggregate and broadcast)
- CSV** Comma Separated Value (File format where columns of data are separated by commas and sometimes semicolons. This format was commonly found in the used datasets)
- HPC** High-Performance Computing (A field of research where homogeneous computer systems so called supercomputers are utilized to address large-scale problems)
- IaaS** Infrastructure as a Service (Cloud computing service layer)
- JSF** Java Server Faces (Web technology in the arena of Java enterprise)
- JSP** Java Server Pages (Web technology available in Java Servlet containers)
- LRS** Learning Record Store (A repository that holds data about learners interactions in a learning system)
- MPI** Message Passing Interface (Standard for implementing parallel algorithms on shared-nothing infrastructures)
- OSN** Online Social Network (An usually web-based online platform where friends, and acquaintances can connect and share information)
- PaaS** Platform as a Service (Cloud computing service layer)
- PLN** Personal Learning Network (In learning analytics describes an individuals interaction with other actors and learning artefacts)
- SaaS** Software as a Service (Cloud computing service layer)
- UML** Unified Modeling Language
- URL** Uniform Resource Locator
- VM** Virtual Machine
- WAR** Web Application Archive

List of Figures

1.1	Dynamic Graph with typical changes	5
1.2	Visual representation of a temporal graph	6
1.3	Strongly connected components in a directed graph	11
1.4	Zachary’s karate club first level communities	13
2.1	Dynamic graph with two edge delete operations	33
2.2	A graph with intermediate, temporal vertices between edges	35
2.3	A multi-graph with time attributed edges	36
3.1	Splits in Structural Graph Partitioning	50
3.2	Schematic of Compute Aggregate Broadcast Computation	56
3.3	Basic building blocks and their interplay	58
3.4	Global State Machine	62
3.5	Vertex Local State Machine	63
3.6	Schematic of an example algorithm run	64
3.7	Four iterations of the label propagation executed over a simple graph with clear community structure	75
4.1	Architectural Overview of the Reference Implementation	86
5.1	Typical architecture of an application using <i>DynamoGraph</i>	111
5.2	A hairball that provides perhaps a general overview but no details [83] .	114
5.3	Temporal snapshots of an email database visualised using the 2.5D method [46]	115
5.4	Calls and SMS from the MIT Reality Commons Network: Friends and Family [3]	116
5.5	IRC Network with computed communities	117
5.6	The namespaces view shows all datasets currently loaded on the cluster	118
5.7	All currently configured datasources	119
5.8	Log of the last algorithms scheduled on the cluster	120
5.9	DynamoGraph in context with learning systems	130
5.10	General structure of the processing pipeline	131
5.11	Prototype screenshots	132
5.12	Politician relation graphs	132

6.1	Schematic of a typical IaaS cloud installation	141
6.2	Degree distributions for combined, in- and vertex out-degree	147
6.3	Average run times of the PageRank algorithm	148
6.4	Degree distributions for combined, in- and vertex out-degree	150
6.5	Run-times for Click dataset import	151
6.6	Run-times for Click PageRank runs	152
6.7	Degree distribution of the Click dataset (cdegree) compared with the Enron dataset (edegree)	154

List of Tables

1.1	Tabular representation of arcs of a temporal graph	6
2.1	All pair shortest paths for the stree stages given in figure 2.1	34

Chapter 1

Introduction

Graph models are a versatile abstraction for many real world problems. Examples of such problems are the recording and analysis of social networks [114, 35], the representation of protein to protein interaction [18] or the connections in a computer network like the Internet [13]. Current models of and computation on graphs often focus on the storage and processing of static data. Static graph models show static snapshots of real world scenarios as perceived at a certain point in time. Graph database systems provide stable storage systems for linked data. Graph processing systems use these graph databases for reasoning on the stored data.

However, the real world references to graph models in many cases are highly dynamic. The network which is observed in the real world is often under constant change such that static models of the network do not reflect to reality accurately. This especially holds for the examples mentioned before and leads to a situation where the dynamic properties of such linked data cannot be analysed sufficiently with the methods of traditional graph algorithms [55, 115, 126]. Methods from temporal graph algorithms can be used to answer questions regarding dynamic properties of graphs.

Moreover the datasets used in many actually graph processing problems are growing very large in size. With domains like social network analysis and bio-informatics working on datasets that cannot be feasibly processed on regular graph database systems especially if they are running on a single system. Storage space, memory and CPU capacities are the limiting factors which can be mitigated through the use of distributed computing paradigms [52].

This thesis elaborates on a framework called *DynamoGraph* which was specifically designed by the author for storage, in-memory management, and processing over large-scale, temporal graphs. The mechanisms provided by *DynamoGraph* and their computational base concepts are described in great detail, the frameworks practical feasibility is shown by applying the approach in the domains of social network analysis

and web-graph analysis, and finally the scalability is demonstrated by running several experiments on compute clusters of different size.

The remainder of this chapter is structured as follows: In section 1.1 the motivating background behind this thesis is discussed followed by highlighting open problems of large-scale temporal graphs in section ?? . Then section 1.3 introduces preliminary concepts and definitions of temporal and dynamic graphs. Section 1.4 further motivates the need for research by illustrating some exemplary application scenarios for *Dynamo-Graph*. From these several requirements (see section 1.5) are derived which lead to a software architectural solution to these requirements (see section 1.6 Hypothesis). Finally in section 1.7 the contribution of this work to the body of research is outlined and underpinned with a list of publications in 1.8. Finally, in section 1.9 an overall outline for this thesis is presented.

1.1 Motivation

Nurtured through a background of interacting with groups in different scenarios when working as a business process consultant and trainer for diverse topics the author discovered an interest on the behavioral patterns of groups. Especially the different stereotypical roles individuals endue in work group scenarios were in the focus of interest. In a preliminary thesis to this work [102] first experiments with behavioral stereotypes according to Raoul Schindler [98] were conducted. As an outcome of this work two main aspects where discovered to be promising for future research.

(1) It became clear that other interesting things in social networks such as fluctuation between cliques and the change of strategically important hubs in a network over time can be discovered. This is possible once social network analysis moves away from mere static graph analysis as done plentiful in current research to the analysis of dynamic aspects of graphs as found in temporal graph analysis.

(2) The computation of rather simple graph metrics mainly based on vertex degree, path lengths, and flow analysis already imposed a computational overhead on rather small dataset of only a few hundred vertices. This leads to the assumption that tools for social network analysis or network analysis in general when performed on the model of temporal graphs need mechanisms that allow them to scale for significantly larger data set sizes. Only if this requirement can be satisfied future computing systems will be able to use temporal social network as a tool to understand human interaction.

Further, this work is motivated by the current trends of affectionate computing [99] where we are on the brink of computing systems to start interacting with humans in a way that feels very natural to humans. Computers are able to interpret natural language, are able to derive meaning from that, and respond to a users input accordingly [31]. Machines today are able to understand a humans context in very much detail. The physical context (location, time, activity, etc.) can easily be captured by modern computing systems. The social context however is difficult to grasp by computers. Humans have an intrinsic sense for analysing a groups structure and defining their own role and the role of each other in a group. If in future systems computers shall interact with humans and groups of humans naturally it will be necessary that the computing actors in the group understand group structure and the roles of individuals in a similar manner than humans.

Finally, it is believed that similar how we are able to observe the interconnect topology of compute systems [59] and reflect on the performance impact different classes of topologies have on a compute system, we are able to observe social systems and their temporal topology. This would allow us to analyse large social systems such as an enterprise and their workforce and optimise their business processes according to naturally occurring interaction patterns in order to improve performance of certain processes.

1.2 Open Problems in Large Temporal Graph Processing

Large-scale temporal graph processing is a field of research which still has many opportunities for exciting research. As detailed in the following section on preliminaries (1.3) and in the chapter 2 on related work, the foundations of this field are clear. The study of networked systems has a vast history and many aspects (algorithms, behaviours, patterns, application areas, path finding, componentisation, etc.) are very well studied.

Certain aspects of the large-scale temporal graph pose new problems which are still open problems. On a formal level the temporal and dynamic graph have been discussed in literature. Their application oriented realization are still a field of open discussion. In this thesis a possible implementation of temporal graph data-structures as a temporal maps are proposed (see section 3.1).

Based on temporal graph data-structures the actual processing over the data is also a topic under active discussion. For large graphs the problem has been under discussion for many years (i.e. high performance computing community) and has found real world solutions in implementations such as Pregel [73]. The temporal information in

networked data, however, adds a new dimension of complexity to the problem. The time dimension grows at a more predictable rate as opposed to the set of vertices. In this thesis a vertex-centric processing solution to this problem is proposed (section 3.4).

Further more application oriented problems are still open to discussion. Firstly queries on temporal graphs are still challenging. Query languages are yet to be defined. Current research on query formalisation is showing first promising results [78]. Secondly the data format for query results is still a challenging topic. Section 3.8 discusses theory and implementation of using the proposed temporal graph processing framework for query tasks.

This results in the final problem observed in this area, which is the visualisation. In general with large-scale graphs established visualisation techniques for graphs break. The 2D output space provided by computers is not sufficient to fully visualize billion-node graphs. It is clear that depending on the application scenario different view levels (macroscopic, microscopic) need to be provided. The temporal aspect in the graph adds another level of complexity. It is yet to be discussed how to efficiently visualize large-scale, temporal graphs. Graph visualisation is generally beyond the scope of this thesis, nevertheless prototypical implementation of a graph visualisation is provided with this thesis and is briefly discussed in section 5.3. Further, the case studies shown used for evaluation make use of graph visualisation (see section 5.6).

1.3 Preliminaries

In the following some terms commonly used throughout this thesis are defined in the context of related work. This is mainly to avoid confusion with terms that are used ambiguously throughout the field of computer science and to clearly demarcate terms that intuitively can be seen as synonyms.

1.3.1 Graphs and Temporal Graphs

In this work the definitions of dynamic and temporal graphs as explained by [43, 58] are used. The following is a summary of these concepts. Slightly different signs and symbols compared to the original papers are used to avoid ambiguities throughout this work.

Definition 1. *A graph G is a pair (V, E) where V is a finite set of vertices, and E is a finite set of edges of unordered pairs (u, v) with $u \in V$ and $v \in V$. A graph G can*

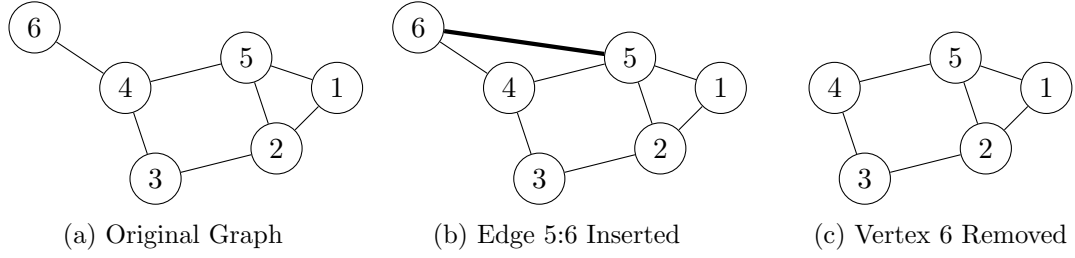


Figure 1.1: Dynamic Graph with typical changes

be called *vertex-dynamic* if the set V varies over time, and *edge-dynamic* if the set E varies over time. Moreover a *dynamic graph* can be both vertex- and edge-dynamic.

This concept reflects to a single static snapshot of a graph which can change over time i.e. edges getting inserted and removed or vertices getting inserted and removed see figure 1.1. The challenge in performing computation on graphs of this type are that that elements (vertices and edges) can change while a potentially long running algorithm is computing metrics on the graph. Research in the area of dynamic graphs mostly covers how to react to these changes and how to adapt computed results given a known set of changes.

Note that definition 1 describes an undirected graph. In a directed graph the same concept of dynamicity holds in the formal definition only the directed graph is comprised of *ordered* pairs. In the directed case the edges are referred to as arcs.

To allow to capture changes in a dynamic graph the concept of a temporal graph was defined.

Definition 2. A temporal graph T is a set of graphs $\{G_0, G_1, G_2, \dots, G_t\}$ where $G_t = (V_t, E_t)$ such that any G_t is a static snapshot of the dynamic graph at time window t in any given time unit.

This means that for any given point t in time the current state of the graph can be determined and moreover static snapshots for a timespan T covering several points in time can be computed. As visible in table 1.1 compared to regular edge lists as found in static graphs edges in a temporal graph also carry a timestamp which refers to the time when the edge occurs.

In definition 2 the set E_t can denote a set of ordered or unordered pairs if a directed or an undirected graph is considered respectively.

By looking at visual representations of temporal graphs as shown in figure 1.2 it becomes clear that the temporal dimension has a strong impact on the network structure. The

Source	Target	Timestamp
A	B	t_1
A	C	t_2
A	E	t_2
E	D	t_3
B	C	t_4
B	D	t_5
D	B	t_6
A	D	t_7

Table 1.1: Tabular representation of arcs of a temporal graph

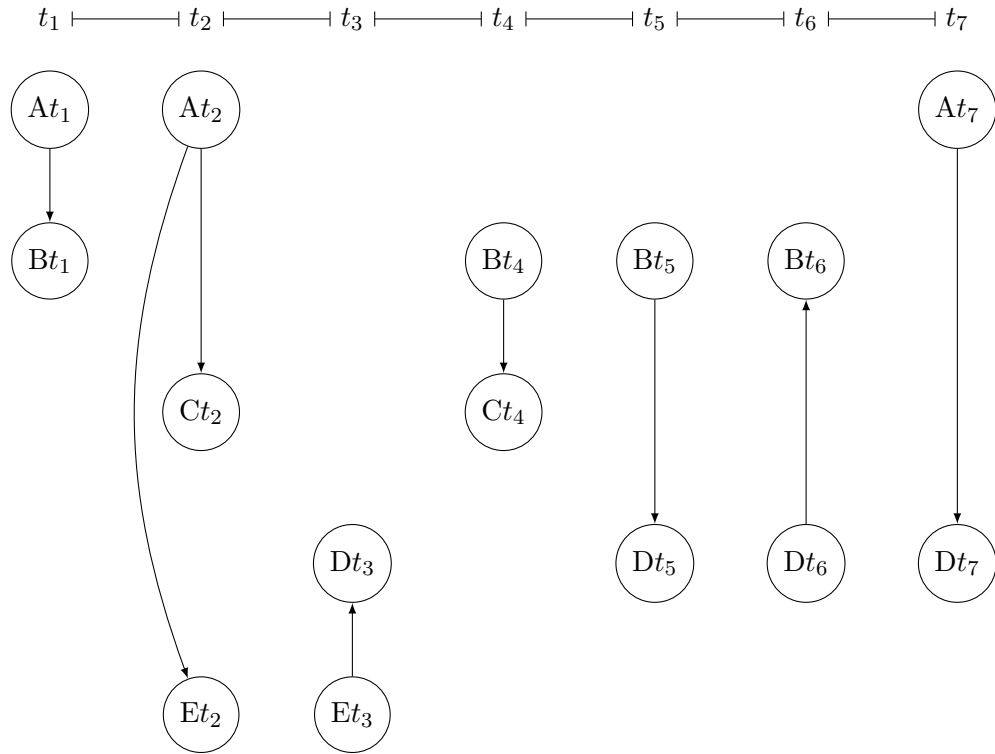


Figure 1.2: Visual representation of a temporal graph

figure shows a temporal graph of 5 vertices and their temporal connections. As clearly visible in the sample graph certain vertices in the graph are in fact only connected at certain points in time. This means that the graph concepts for reachability namely paths and connected components need to be redefined in order to be applicable for temporal graphs.

Definition 3. A temporal path p_{i-j}^h is an ordered set of vertices $v_i \in G_t \in T$ starting at vertex v_i and ending at node v_j . It is defined as a sequence of hops over distinct vertices in a defined time window. The path is allowed to hop between time windows a maximum of h hops.

In this work we elaborate on graph data that is stored and processed on top of a cloud based distributed compute cluster (see section 1.3.4 for preliminaries on that topic). Throughout this thesis the term *vertex* will be used for vertices (nodes) in a graph. In our application domain the analysis of communication data and their corresponding social networks each vertex will represent an individual in an organization. In our implementation we want to address real world problems where also further information that is associated with a vertex needs to be stored. We call this further information the vertex profile or *profile*. It can contain any arbitrary data. Links between vertices are referred to as (directed) *arcs* and (undirected) *edges* throughout this thesis.

1.3.2 Static Metrics and Analysis

Obviously graph models were created with the goal to better understand how certain networks behave in nature. Thus many different graph metrics and analysis algorithms have been developed in the past. These algorithms most often focus on the analysis of static graphs and are currently extended in various novel approaches (see 2.1). This section will highlight the most important metrics in graph analysis.

In general graph algorithms can be grouped in algorithms that perform global analysis i.e. try to find a measure that describes the network as a whole (examples of such metrics are the small-world property [6], and the density of a graph) and algorithms that compute local metrics of vertices or sets of vertices (examples are vertex centrality and vertex degree). Further, algorithms are often categorised into path following and reachability algorithms that cope with the topic of whether and how a certain vertex is able to reach other vertices and into algorithms that try to cluster and partition graphs.

1.3.2.1 Centrality

The term centrality refers to the process of identifying the most important vertices in a network. However, importance is a fact that needs to be determined differently according to concrete application scenarios. Thus resulting in different metrics for centrality being established. In an online social network one might assume that the vertex with the highest number of neighbors (i.e. most friends) is the most popular person. This is known as degree centrality in network analysis. Another definition of popularity might be that the person that is able to reach most people in the network quickly is most important (closeness centrality), and a further definition can be that the person that is a relay on most communication channels is most important (betweenness centrality).

1.3.2.1.1 Degree Centrality

One of the simplest local metrics in graph theory is the degree of a vertex: N_i . It measures the number of neighbours of a vertex where $N_i = \sum_{j \in V} e_{ij}, e_{ij} \in E$. The degree is defined for each node (though slightly different definitions are to be found for weighted and directed graphs) and can be used as a measure for popularity. The degree centrality of a vertex is defined as the vertex degree normalised by the maximum vertex degree $N = \max(\{N_i, N_{i+1}, N_n\})$ in the graph or more formally:

$$C_i = \frac{N_i}{N - 1} \quad (1.1)$$

The definitions given above refer to a measure of popularity in undirected graphs. Its lower bound is $C_{min} = 0$ and its upper bound is $C_{max} = \frac{N}{N-1}$. Degree centrality can also be computed for directed graphs and are then known as in-degree centrality and out-degree centrality. In-degree centrality is also often referred to as prestige in literature i.e. a paper which was cited by many others will have high in-degree centrality in a citation network and thus have high prestige, or a person nominated by many others in a social network for an award will have high prestige.

1.3.2.1.2 Closeness Centrality

Another definition of centrality is the fact of how swiftly a vertex can transport information to all other vertices in a network. In other words how close the vertex is to all other vertices on the network. Closeness centrality is computed as the average shortest path between a vertex and all other vertices in a network. With n denoting the number of vertices in the graph and d_{ij} denotes the length of the shortest paths between i and j .

$$C_i = \frac{1}{n - 1} \sum_{j \neq i \in V} d_{ij} \quad (1.2)$$

In social network analysis individuals who are highly connected within their own cluster or community show high closeness centrality ($C_{max} = 1, C_{min} = 0$). This metric is important in many applications since high closeness centrality often refers to influential positions in a local community. In the social network of a corporation individual positions with high closeness centrality are often not known to a broader public in the company but are respected in their local community and are able to quickly spread information in their local community.

1.3.2.1.3 Betweenness Centrality

From an applications point of view the metric of betweenness centrality can be seen as the opposed metric to closeness centrality. Betweenness refers to vertices in a network that are sitting between clusters and communities and thus are able to bridge information in a communication or social network. This metric is very important in social network analysis for instance where this metric marks information brokers that are able to transport information between communities which otherwise would not communicate or not communicate efficiently. Real world examples on general social networks are quite naturally actual brokers (real estate, mortgage, etc.) which make a living by connecting otherwise disconnected groups.

The betweenness centrality of a vertex is computed by counting the all pairs shortest paths between vertices in a network, that pass through this particular vertex. In the following given for a directed graph.

$$C_i = \sum_{j \neq i, k \neq i \in V} \frac{p_{jk}(i)}{p_{jk}} \quad (1.3)$$

A bridge in an otherwise disconnected network will reach maximal betweenness in that network ($C_{max} = 1$). Whereas nomadic vertices that are connected only through a single arc or edge to the network will have minimal betweenness centrality ($C_{min} = 0$).

1.3.2.2 Paths and Reachability

A further class of algorithms over networks are reachability metrics and more general path finding algorithms. Reachability allows to ask the question, if vertex $b \in V$ can be reached by following edges starting at vertex $a \in V$. Any found route between a and b is called a path. In general two naive approaches for path finding and thus confirming if any two vertices have reachability are known. Naive path following can be conducted as a breath first search strategy or in a depth first search strategy. The latter is easier to implement since recursive function calls can be used instead of maintaining a data-structure of already visited paths. Both strategies have bad time complexities of $O(|V| + |E|)$, which for very dense graphs is $O(|V|^2)$.

In practice significantly better performance is reached for instance by using heuristics in the search process. Especially in application scenarios where a certain sense of proximity and distance can be computed from general vertex attributes (i.e. locations on a road

map) the a^* class of algorithms which is based on Dijkstra's famous search algorithm are used. A^* search runs in $O(b^d)$ time complexity (b denoting the branching factor, and d is the worst case depth of the solution). So in general it relies on the quality of the heuristic function that prunes target nodes of too high cost for traversal.

Paths, shortest paths, and reachability are very well studied over static graphs. In temporal graphs reachability needs to be redefined because reachability will also depend on the time or frame of time a network is observed in. Certain paths will exist only at certain points in time.

1.3.3 Important Subgraphs

Many problems in network analysis relate to parts of a larger graph. These parts are also often called partitions, clusters, and communities usually depending on the exact area of research. In general a subgraph is always a set of vertices and edges which themselves are an interesting subject to study without the necessity to have a global view of the complete network.

The task of extracting subgraphs from a network is often motivated from an analysis task where parts of a network are to be analysed in greater detail. From an applications point of view it is mostly desirable to extract subgraphs that conform to natural understanding of networks i.e. social networks. Such that subgraphs are selected that reflect groups and communities that occur in natural networks. The terms *community* and *group* refer to the problem of extracting subgraphs from a network that are naturally understood as groups by a human observer i.e. communities found in social networks.

From a pure technical point of view the problem of splitting a graph into individual partitions is motivated by the fact that very large graphs cannot be feasibly handled by the working memory of a single computer. Thus the term *(graph)-partitioning* refers to the problem of splitting a graph of arbitrary size over a set of given resources i.e. number of partitions.

1.3.3.1 Connected Components

The most strict of a subgraph forming a community is the *connected component* (or sometimes referred to as just component). A connected component is a subgraph in which any two vertices are *connected* to each other by paths. Furthermore a connected component is not connected to any other vertices in the supergraph.

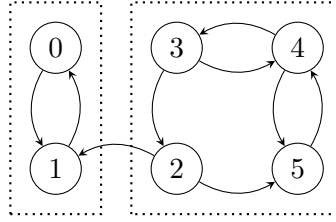


Figure 1.3: Strongly connected components in a directed graph

Natural data observed in typical application areas such as social networks shows that these networks (after the forming phase) are most likely completely connected. From any vertex selected from the network at random any other vertex in the network can be reached by tracing edges between vertices. This means that the whole network forms a single connected component which is called a giant connected component or just *giant component*. While from an analysis approach this property in a graphs does not give intuitive interpretations of a network. However, it is important to note that in certain domains all natural graphs expose this giant component property which is an important factor when designing algorithms such as the Erdos-Reney random graph algorithm [30], which create artificial networks closely resembling properties of actual networks.

When observing directed graphs a slightly different definition of connected components can be given. A subgraph is denoted as a *strongly connected component* if every vertex in the component is reachable from every other vertex in the component by paths which strictly obey edge direction. Figure 1.3 displays a directed graph which is not strongly connected. The shown graph however has two strongly connected components which are marked with dotted lines.

Strongly connected components can intuitively be interpreted as strongly connected communities in a network. It is very likely that information spreads very fast in strongly connected communities.

1.3.3.2 Clusters and Communities

In many real world scenarios however the concept of strongly connected components is too strict to reason over clusters of communities. Clusters and communities are common in real networks. In social networks for instance communities form based on common interests, location, profession, etc. This is true also for other networks such as citation networks where communities are formed by research topic or metabolic networks which have clusters of functional groupings. In a naive approach obviously clusters and communities can be found following vertex attributes i.e. clustering all

vertices in a network which show a certain property such as the same job in a social network.

However, if such attributes are not available in the dataset or their existence is to be verified clustering algorithms that are computed over the topology of a graph. Besides many other algorithms and possible variations of them the Girvan-Newman method is one of the most popular algorithms for finding communities. The fundamental property that quantifies communities in graphs is the clustering coefficient [34]. It is defined as (where N_t denotes the number of triangles on the graph and N_c the number of connected triples of vertices):

$$C = \frac{3 * N_t}{N_c} \quad (1.4)$$

The clustering coefficient describes the probability that two neighbours u, v of vertex x are also neighbours themselves. In a fully connected graph (every vertex is connected with every other vertex) the clustering coefficient is 1. Typical values for real world networks are in the range of 0.1 to 0.5.

This method is based on the betweenness metric which can be used to find the vertices living on the edge of clusters. Instead computing betweenness for vertices it can also be computed for edges as in counting the number of shortest paths that need to pass through a certain edge. The assumption in the Girvan-Newman algorithm now is that in highly clustered networks the edges which are between clusters will show very high betweenness and edges inside of well connected clusters will show low betweenness. The algorithm is now able to find community structure by iteratively removing the edge with highest betweenness from the network and re-computing betweenness afterwards. Through this process edges connecting individual clusters are removed such that the network unfolds into connected components only containing the vertices that belong to a certain community.

Figure 1.4 shows the Zachary's karate club social network [124]. For this network of a real university karate club it is well known that the club was split into two groups at some point in time. In the figure the vertices are drawn as rectangles for one group and as circles for the other group as assigned through a run of the Girvan-Newman algorithm. Compared with data from a questionnaire available for the original dataset only the vertex with id 2 was labeled incorrectly. The network was manually arranged for better readability. Considering no additional data is available it is also difficult for a human observer to determine which group to assign vertex 2 to.

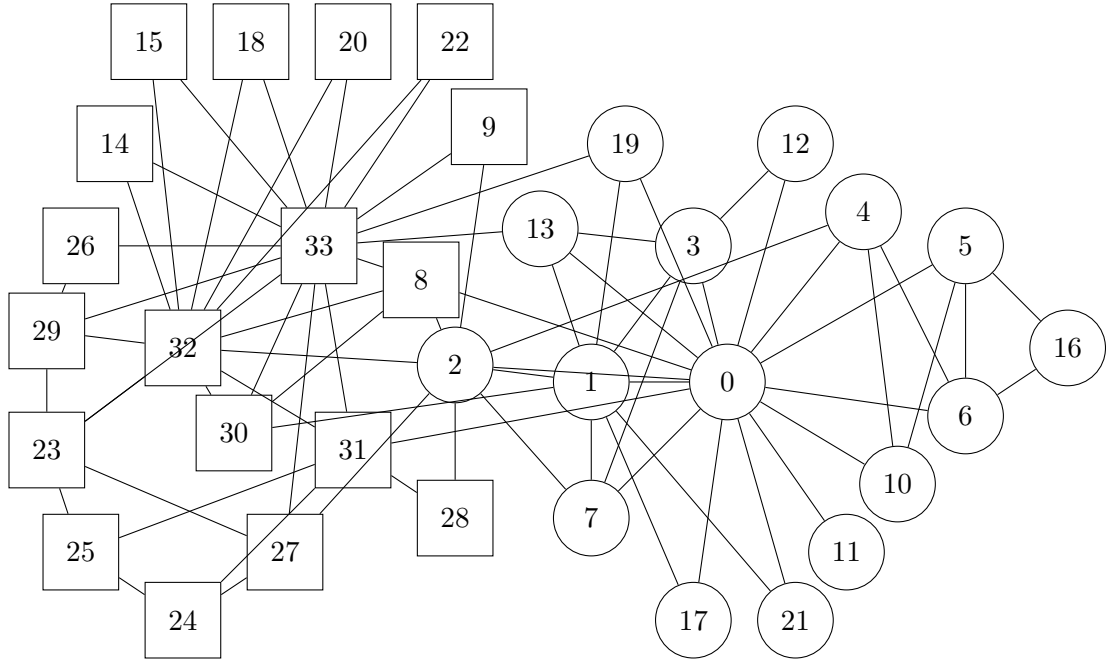


Figure 1.4: Zachary's karate club first level communities

Obviously there exist many more algorithms for graph clustering which were developed from different background and thus use different approaches and parameters to compute optimal communities. The decision to use a certain approach most often depends on two factors. First whether a certain approach was designed for a certain application scenario such the Girvan-Newman methods have shown to be applicable in social network analysis and depending on the parameters used will cluster a given social network into communities which are perceived in a similar way by humans. On the other hand the algorithms also need to be feasible for the problem size. Both the computational complexity and memory complexity have limitations which when broken make the application of graph clustering infeasible [97].

Graph clustering can be categorized into global and local clustering algorithms. Global algorithms compute clusters for a complete graph topology and thus require the complete dataset to fit into memory which in the case of a dense graph will result in memory complexity of $O(N^2)$. Existing global approaches are capable of dealing with up to a few million of vertices on sparse graphs [47]. For large graphs computation becomes challenging which is typically also the case for graph clustering. The runtime complexity of clustering algorithms should stay near the boundary $O(n * k)$ in order to be scalable, sub-linear complexities would be highly preferred. Traditional clustering algorithms such as the Girvan-Newman approach [85] are based on betweenness centrality. In iterations the weakest ties (the edges with highest betweenness centrality) are removed from the graph. Betweenness requires the set of all shortest paths as input, the fastest known exact approach is Brandes [14] with a runtime complexity of $O(nm + n^2 \log n)$. Thus these algorithms are inadequate in the case of very large graphs.

An algorithm suitable for large scale networks was developed by Raghavan et al. [94]. It is a local algorithm which is based on label propagation. Assuming that a vertex v_0 has neighbours $v_1, v_2, v_3, \dots, v_n$ then v and each of these vertices has a label denoting the community they belong to. Vertex v can determine its own community by observing the community labels of its neighbors. Initially each vertex lives in its own community, through an iterative process any vertex v updates its community label by observing its neighbors community labels and setting its label to the community label which is most often found (perhaps considering edge and node weights) amongst its neighbors. In sparse graphs densely connected groups quickly reach consensus on a unique community label. Assuming that k (the number of average edges between vertices) is low and thus can be neglected and most communities reach consensus after an also low maximum of t iterations of the process the algorithm presented can assign communities in near linear time ($O(n)$).

Clusters and community in networks are a topic which is obviously not only addressed from a pure technical perspective but also has a wide body of research from sociological [81], psychological [98], and societal research. From the viewpoint of these disciplines questions on how and why communities form, how and why they interact and whether or not they are able to follow a common goal are in the focus of research.

In general there are two terms which are important to discriminate in the discussion: community and group. These two terms from a sociologists point of view refer to groups of people and can be distinguished in that sense that communities describe a set of people who are interacting in some sense, such as the community which is formed by the set of people working at the same office and thus are interacting. Whereas a group not necessarily has to have interaction, a group for instance could just form from a common describing attribute such as the set of people who hold a masters degree in statistics [112].

1.3.3.3 Graph Partitioning

The problem of finding splits in graphs has obviously been addressed long before the work of Girvan and Newman. However, the point of view for looking at this problem has been slightly different. Where clustering and community detection aims to find natural clusters in a network the algorithm group of partitioning algorithms tries to find a minimum cut in networks. This means that a network is to be split in a pre-defined number of partitions (usually of similar size). The partitions are found by trying to minimize the number of edges that are spanning the split.

This way of graph partitioning is applied in scenarios where technical aspects of networks are in the focus. For instance in a distributed computing cluster where the communication structure of a certain algorithm is known these minimum-cut algorithms can be used to determine optimal schemas for the load-balancer to distribute the workload. Obviously this method is less ideal to be applied on general networks because regardless of the implicit community structure in a network a minimum-cut algorithm will always find a fixed number of communities.

1.3.4 Cloud Computing and Big Data

Cloud computing is the state of the art computing paradigm to build current web scale computing architectures. The implementation part of this thesis is largely built on top of cloud computing infrastructure.

In general cloud computing is defined as a *model model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction* [76].

Cloud computing is defined on three service layers: Infrastructure as a Service, Platform as a Service, and Software as a Service, each with very specific characteristics.

Infrastructure as a Service (IaaS) provides virtual computing infrastructure (servers, network, storage, etc.) that can be provisioned through software interfaces. The consumer of IaaS resources is decoupled from implementation details such as concrete server hardware and their setup and maintenance but has full control of virtual compute resources starting at the operating system level. In IaaS computing resources are often called a node, compute node, and virtual machine. To avoid possible confusion of the term node with vertices in a graph the term node is only used for compute nodes.

On top of the IaaS tier Platform as a Service (PaaS) is living. PaaS provides computing services in a defined environment. A platform consists of a runtime environment that is capable of executing payload code and usually supports a defined set of programming languages. The platform most often provides a defined set of function libraries that provide extended functionality over what is available by default for a certain programming language, and finally platforms most often support for data storage, message passing, caching, log file storage etc. All components of such a platform are designed in a way that they can be used in a highly scalable manner.

PaaS deliberates its users from aspects such as software updates, operating system maintenance tasks, and any mundane tasks related to scaling systems up and down. This allows for software architectures of automatically scaling systems. In the context of this thesis the reference implementation is built in a manner that it can be run on top of IaaS clouds and is itself provided in the PaaS model.

Users of the system described in this thesis in turn will most likely provide their applications in the Software as a Service (SaaS) model. A model where the user of a system is accessing the service through a web browser and is completely freed from any system maintenance related tasks such as software updates and backups. In the context of this thesis the demonstrator applications fall in the category of SaaS.

Another field this thesis touches is the field of big data processing. Big data discusses the problem of data that grows so big in size that it becomes awkward to handle on traditional systems [49, 71]. The data becomes big in three dimensions often referred to as the 3V's (1) volume, (2) velocity, and (3) variety of data. For the volume and velocity dimension the requirement of processing this data on highly scalable systems that run on large scale compute clusters becomes apparent.

1.4 Addressed Application Areas

The framework presented in this thesis is a general computational framework for processing over large scale temporal graphs. As described in the introduction automated social network analysis is a perfect match as the main application scenario. In the following section concrete application examples from automated social network analysis are described in greater detail. Since the framework is designed to be generic, other application scenarios from other fields of research such as the analysis of computer networks are also described in this section.

1.4.1 Social Network Analysis

In the recent years analysis of social networks has seen a big hype. Many publications are discussing different topics in this field of research (organisational issues [125], size of networks [87, 26, 121], etc.). When reviewing literature it is important to clearly distinguish the different forms of social networks available today. In literature and popular news the term social network is often used synonymously with the term online social network (OSN). An online social network is a web-based system where users can register a profile often with their real name and have the ability to connect with friends

and acquaintances through creating a mutual connection often referred to as friend request. On top of this basic functionality OSN provide tools for sharing information and media with peer users. Depending on the topic of main interest we find famous representatives of such OSN in Facebook, Google+ and VK.com for general friendship networks, and LinkedIn and Xing for professional networks.

In contrast to that Sociology refers to a social networks when discussing about networks that are formed between humans. These networks do not only exist in the digital world but have always formed in real life and have been studied under various aspects such as in the field of Sociometry [81]. This means that not for all social networks digital representations are available such that these need to be derived through other means from interaction, communication and linked data [109, 2]. During this process it is important to also analyse typical characteristics of social networks. General networks can form many-fold, such as found in e-mail communication where people are interacting with each other. On other media however formed networks do not necessarily form social networks and thus communities. An example for this are blogging and microblogging (i.e. Twitter) services. In these systems users can follow other users news and messages and interact on this content. Which means that the original creator of content is not necessarily aware of the audience he or she reaches with it. This awareness occurs only during interaction and the community forming phase; when another member of the system reacts to content effectively creating what is called mutual awareness [69].

On top of social network data many-fold analysis can be performed depending of the goals of a study. In the past a vast amount of literature discussed social network analysis from many different points of view. In the following some of these studies are picked to highlight application areas that could benefit from temporal graph analysis.

1.4.1.1 Sizes of Social Networks

In their famous study of the small-world property [121] that is inhabitant to many social networks Watts and Strogatz describe a property that from any starting vertex in a social network any other vertex in the network can be reached through a maximum of N hops. This ground-breaking work was later detailed in follow up publications [120, 119]. Already from these studies it is clear that a property such as the small-world phenomenon is interesting to study in the context of a temporal network. Static networks such as those described by online social networks have been used to verify the property. The question however is if the information stored in an online social network is inaccurate in that sense that the temporal aspects of a real social network (vertices appearing and vanishing) are not addressed by the model. Temporal networks observed in particular short time-intervals might exhibit that the general reachability assumption

(any social network forming a giant connected component eventually) is not holding. A temporal network will rather show multiple smaller connected components [11].

This is also underpinned by work of Amaral [6] that (besides other interesting properties) shows that clusters in real social networks saturate in that sense that while communities and clusters grow the probability of new vertices joining a community shrinks. Depending on the network it seems that there are some limitations for the size of clusters. It is assumed that the death of vertices and thus their eviction from clusters creates new room for further vertices joining the cluster. This model is detailed from a different point of view by Dunbar [87, 26] who claimed that there is a cognitive limit as of how many social contacts humans are able to keep track of. This property is referred to as Dunbar's-Number in literature and is claimed to refer to a social group of around 150 persons that one can keep sensible relationships with. The property has been verified in online social networks [36]. Temporal networks could help in this area of research in providing a temporal model of large social networks and allowing researchers to study the small-world property and Dunbar-Number in sliding windows.

1.4.1.2 Communication Networks and Group Performance

Communication networks in the age of digital communication leave communication traces in e-mail databases [20], chat logs, and phone logs. All of which highly accessible to analysis tools. Communication networks allow for a multitude of aspects to be analysed. For instance various publications on graph clustering [97] use them for automated group detection. On a smaller level communication structures and considering the temporal aspect communication patterns can be observed. In the past studies analysed the effect of communication structures on group performance for instance [67]. This allows to observe whether or not certain structures lend themselves better to solve collective tasks than others.

In temporal graph analytics these approaches can be extended to predict whether certain groups are improving in terms of communication structure compared to their past. Set aside concerns regarding privacy, organisations could monitor their structures to detect communication deficiencies and bottlenecks.

1.4.1.3 Co-Author and Co-Actor Networks

Another important class of social networks are co-author and co-actor networks. These classes of co-* networks have also sparked popular interest mainly because their data

can easily be understood by a wide audience. This is especially true for the movie co-actor network. Actors for instance are trying to be as close in the co-actor graph to other famous actors as possible. There exists a parlour game which is referred to as the *six degrees of Kevin Bacon*¹. The goal of this game is given any actor to find the shortest path between Kevin Bacon and said actor. It is believed that Kevin Bacon acted in so many movies and TV-productions such that every film-actor can be linked to Kevin Bacon in a maximum path length of 6 in the co-actor network. The co-actor network is manifested in various databases. One of the most popular ones being the Internet Movie Database (IMDB)² which besides pure movie and TV-show metadata also provides crowd-sourced ratings. From the IMDB data it has been shown that the actor relationships in fact form a giant connected component [44] such that the assumption that every actor with significant public exposure cannot be farther than 6-degrees from Kevin Bacon.

In science a similar phenomenon can be observed with the co-author network. Co-author networks are the dataset of choice for many graph and network related publications. This is mainly due to the availability of data in various research paper databases and again that researchers are well aware of the data inherent in the co-author networks. In some scientific communities the author Paul Erdős is the undisputed center of earth³. This is at least true for mathematics and related sciences. Paul Erdős was a very active author and engaged with more than 500 collaborators. This is why it has become some sort of sport to reach a very low Erdős-Number as being as close in the co-author network to Erdős as possible⁴.

Besides their popularity and the reachability games played on top of them, both networks (co-author and co-actor) also provide temporal information. In co-author relationship the publication date marks a point in time when the authors have collaborated and for movies this is given by the release date.

1.4.1.4 Criminal Networks

A class of networks that could also benefit from temporal graph analytics are criminal networks. Criminal networks are a tool in criminal investigation. Oftentimes the mere visualisation of actors in a case helps to shed further light onto a situation. This tool is also well known from popular fiction where the actors, their pictures and relationships are pinned on a board for improved case overview. In the real world these kind of

¹SixDegrees.org <http://www.sixdegrees.org>

²Internet Movie Database (IMDB): <http://imdb.com/>

³Paul Erdős Project: <http://www.acs.oakland.edu/grossman/erdoshp.html>

⁴As a side note: Paul Erdős was also active in graph and network theory. Many of the papers cited in this thesis are authored by people with particular low Erdős numbers around 3 to 4. Also the thesis advisors have spectacular low Erdős numbers (Paul Spirakis: 3, Gabriele Anderst-Kotsis: 4).

information are stored in information systems which enable automated network analysis or data is pulled from archive in cases where large conspiracy networks are considered [8].

Also for this class of networks temporal aspects are of uttermost importance. For instance in a current case and the related network of actors it can be of importance to compare with past networks to find hidden nodes and thus other actors that might also be involved a current case [17]. In naive analysis approaches one can assume that actors that show up as actors in multiple past networks have a higher probability of being involved in a current case.

1.4.1.5 Learning Networks

Social interactions also play a key role in the process of learning. Modern approaches for learning are showing tremendous shifts in learning methods away from a teacher to learner information flow in static classroom settings towards mentored self-driven approaches. The latter have been enabled by various technological advancements that now make digital learning spaces a reality. However, in modern learning scenarios also roles of the actors are breaking up. Individuals can occupy multiple roles such as mentor, trainer, trainee, mediator, information-hub, etc. even in parallel.

This leaves room for new research regarding the analysis of what is called the learning network. Meaning the network that actors involved in learning and they artefacts they use in the process. Temporal graph analytics can help analyse how individuals change subjects as they progress through their personal learning experience. Temporal graph clustering can help to show how individuals addressed different topics over time. This information can be used by other learners to identify efficient learning paths.

1.4.1.6 Political Networks

Another class of interesting networks are political networks. These form through the interactions of politicians and political parties. While parts of the political network are not visible to the general public the official political discussions of legislative bodies such as parliaments of the house of representatives in various countries are well documented, even throughout history, in transcripts. Many countries worldwide are following the Open Data trends and as measure of transparency these transcripts are made available to the public. In some particular cases these transcripts are even annotated and converted into structured data such that the political debate can be converted to a network diagram with only little overhead. This is indeed the case for example for

the U.S. House of Representatives [91] and the Italian Parliament [7]. Which naturally have been the subject of studies.

Political networks are very interesting to observe over time to get a sense how the political landscape evolves over time. This can be done at a very macroscopic level [91] as in observing how different political forces evolve over time. In the example of the U.S. government, it can be observed that the two strongest forces in the country have been drifting apart over time. While political analysts might have observed this through the use of temporal graph analysis this property can be underpinned with data. It comes with no surprise that an online piece which was published on the blog of Renzo Lucioni in 2013⁵ covering the senate voting relationships received much press coverage.

On a micro-level the individual clusters within a governing body can be studied. An interesting property of these clusters is the inner cohesion of these clusters. The work of Amelio [7] gives room to the assumption that temporal analysis of political networks and especially the evolution of cohesion can be used as indicator for how successful parts of a political body work, and consequently can be used as predictors for whether or not a certain political group is going to be (re-)elected as the government.

Inspired by the aforementioned work also the framework described in the presented thesis was used as basis for political network analysis. Using the historical data from the Austrian parliament it was shown that the temporal graph can be used to reason over coalition and opposition and their evolution over time [106]. Details of this case study can be found in section 5.6.2.

1.4.2 Computer Networks and the Web

Since the beginning of the Internet small disconnected local area networks have merged into one giant connected network referred to as the Web. While the perspective leaves out the fact that certain network segments are strongly firewalled from the rest of the network it is still safe to assume that large fractions of the modern computing infrastructure are connected to the Internet. This trend is even moving on to connecting even more user devices than ever which is currently being coined as the *Internet of Things*.

The Web in its more than 25 years of history has transitioned through many phases. From a network for education and research it has constantly picked up economic trends such that it has reached its commercial peak by now. The individual phases of the Web

⁵Renzo Lucioni: <http://www.renzolucioni.com/senate-voting-relationships/>

and the Internet must also be visible in the network structure itself. It is safe to assume that in the beginnings only a very little number of nodes was connected to the network and certain hotspots hosted much of the information available on the Web. This trend moved to a very distributed landscape of nodes with certain nodes on the web being of higher importance than others (see rating algorithms such as PageRank [89]) and generally a larger audience of consumers using the network. Today the landscape has again shifted towards, on the one hand a distributed set of nodes that has already led to the situation that on a global scale we ran out of IP addresses, but on the other hand content provided on the Internet being mainly driven from what is called information silos. Social media sites such as Facebook, Twitter, and LinkedIn but also popular search engines and information portals such as Google account for such silos. Content is not created on distributed nodes but in only a small number of silos.

From a temporal aspect the different phases of the Web can be observed through the use of temporal graph analytics. Moreover real-time usage information about the web can be analysed to see shifts in graph metrics over time. It can be used to observe how certain sites on the web become popular and others drop in popularity and go out of service eventually. A foretaste of such temporal network analytics is given in chapter 6, section 6.4.2 where the system developed for this thesis is evaluated over a real-world click graph.

1.4.3 Cartography and Maps

The final class of temporal networks addressed in this thesis are maps. Digital cartography lends itself to graphs as data-structures as a model for road networks. These have been used for many decades and some of the most important graph algorithms originated as problems over road networks. Although these may be outperformed by faster derivatives [24] some algorithms such as A*-search are still an important base model for heuristic path finding in graphs.

Road networks are also undergoing constant change. Compared to the other networks listed in this section they naturally change at a much slower rate. This might lead to the assumption that a temporal data warehouse for road maps is not required. However, for modern road networks up to date usage statistics from road-side sensors and mobile phones are available. Such that road networks can be used to reason on the mobility patterns of individuals [38]. This together with temporal graph models allow to study the effects of short term changes (traffic blocks) and long term changes on the road network on usage statistics. This can be used in traffic planning to simulate various scenarios for future projects.

1.5 Requirements

From the last sections requirements on a system suitable to process large-scale temporal graphs efficiently can be derived. First of all a system like this needs to be able to store temporal graph data and needs to provide fundamental support for operating on the temporal dimension of the graph. Further the system needs to be scalable in terms of allowing to handle graphs of a very large scale. Finally as a result of the scalability requirement special focus needs to be put on fault tolerance since fault in very large systems is more likely and needs to be tolerated to some extent by the system.

1.5.1 Temporal Dimension

The temporal dimension of networked is a key criteria for a system for temporal graph processing. This means that the systems users need to be able to store a temporal graph in snapshots as proposed in [43] and [58]. These snapshots will refer to static views of the graph for certain points in time. This will allow for any point in time for the set of current vertices in the graph and the set of edges connecting these graphs can change. Moreover other properties of vertices and edges also need to be stored with temporal indexing such that any attribute of an object in the graph can change over time i.e. vertex and edge labels, metrics, and descriptive attributes.

Users of such a system need to be able to use the temporal dimension of the graph in queries and processing tasks. It needs to be possible to generate a static snapshot of the graph for a certain period of time such that static graph metrics can be computed as in traditional graphs. Further it needs to be possible to directly address the temporal components of the graph such that temporal properties such as the temporal distance [115] between any two nodes can be computed.

1.5.2 Scalability

Further such a system in order to be useful in practice needs to be able to scale to larger graph sizes. From the nature of natural temporal graphs such as social networks we get the impression that their change in data size is mainly growth. If one observes a social network described by interaction it becomes apparent that the number of edges in the network will grow over time. Special mechanisms would be required to make the system forget older data that is perhaps not relevant to current analysis tasks. On the other hand also the number of vertices seen in the network might fluctuate. Vertices will most likely be added to a network and only very seldom be removed.

A requirement will be that the system can be amplified and narrowed according to current resource demands. It has to be possible to perform these changes in system size during system runtime without the need to stop the system and reload a possibly very large dataset from backend storage.

1.5.3 Fault Tolerance

Finally a system handling large-scale networked data needs provide fault tolerance capabilities. As systems grow larger in size also the probability that any of its component fails increases. Such that as a system designer one needs to accept the fact that one or more components of the system will fail eventually.

When thinking about application scenarios where temporal network data is collected and stored in a processing platform failure might not be acceptable. Especially if the collected data represents a valuable asset. This means that the proposed system needs to be fault tolerant and be able to recover from failures of part of the system automatically.

1.6 Hypothesis

As outlined in chapter 1 interesting and important questions on networked data currently cannot be answered, because the temporal dimension of such data is widely neglected in frameworks for large-scale graph processing. It is assumed that these open issues can be tackled by providing mechanisms for temporal graph partitioning, which allow distributed storage of temporal graphs, and the application of methods from distributed computing to be applied to perform distributed temporal graph processing.

1.6.1 Temporal Graph Partitioning

On the scalability dimension of the problem the hypothesis is that by using cloud based infrastructure very large scale network datasets can be partitioned in such a way that certain computing and storage instances are responsible only for parts of the graph. This problem is already a known problem discussed in work related to graph partitioning.

As explained earlier the definition of temporal graphs is a set of static graphs which reflect the snapshots of the temporal graph at certain time-windows. It would feel very

natural to partition graphs along the temporal dimension since this would go conform with their definition. This is called a multi-slice temporal graph model [82].

However, it is very likely that the growth of a temporal graph along its number of vertices outnumbers the growth in the temporal dimension. In this case it might be better to store the graph as a model with time-stamped edge lists [55] and structurally partition the graph. Which is done in the same fashion as found in classical graph partitioning algorithms. This way of partitioning will most likely integrate better with existing graph processing systems.

1.6.2 Distributed Temporal Graph Storage

Once a feasible partitioning scheme for temporal graphs can be found the next problem of distributed temporal graph storage needs to be solved. Distributed storage of data poses the problem that elements of a dataset that span several partitions are usually difficult to split. In the scenario of linked temporal data and under the assumption that time-stamped edge lists can be used as the model a vertex-centric approach of representing the data can be chosen.

The assumption is that a vertex in a graph can be stored in a fully self-contained fashion such that each vertex carries all the information about itself. This means that all meta-data for a vertex (name, and any other arbitrary attributes) are stored together with time-stamped edge-lists of the vertex in a document.

An edge can only exist if both the connected vertices of the edge exist such that it is a valid option to avoid storing edge-lists independently of vertices. The only downside to this approach will be that data about edges will be duplicated.

1.6.3 Distributed Temporal Graph Processing

In recent research on distributed computing a trend for processing large-scale datasets can be observed. When methods of distributed computing are applied most often mechanisms are used where computational intelligence (i.e. code) is sent to the data for local processing and aggregated results are sent to a central coordination instance. This allows for better scalability of the computing systems and avoids bottle-necks imposed by network links.

It is believed that by applying compute aggregate broadcast mechanisms [63] and more precisely a Pregel style processing model [73] on top of the distributed temporal storage also efficient distributed processing can be implemented.

1.7 Contribution

Assuming that feasible, scalable, and stable implementations for the problems mentioned in the last sections can be found the contribution of this work is a framework that closes a gap between theoretical work in the area of temporal graph algorithms and large-scale graph processing. A software package like the presented can make temporal-graph algorithms applicable to natural datasets and thus will enable further research on the dynamic properties of large networked systems.

The research for *DynamoGraph* phased through several milestones, each of which providing results that are documented in this thesis. In a first phase a general literature review over the current body of dynamic graph algorithms, temporal graph models, large-scale graph problems, and distributed computing was conducted. The outcome of these is committed to paper in a previous section on preliminaries (see 1.3) and in greater detail in chapter 2 on related work.

Parallel to literature review first application scenarios i.e. in the analysis of behavioural stereotypes in social networks were analysed and published as a first conference paper [109]. This clearly motivated the need for a distributed computing approach since it was not possible to compute complex graph metrics in feasible time. This resulted in distributed storage concepts (distributed temporal maps) for temporal graphs which are documented in this thesis in great detail in chapter 3, section 3.1. Based on the temporal maps first prototypes of a distributed computing platform on top of existing Big Data solutions were built. These first prototypes showed undesired behaviour on top of certain data patterns (directed temporal graphs with loops) such that details about them are not part of this thesis but are documented in the following paper: [110].

From literature review and overall technological decisions a set of possible graph partitioning concepts were at choice. Graph partitioning is a very well studied topic such that the contribution of this thesis is merely a discussion of graph partitioning strategies applicable for large-scale, temporal graphs. Properties of graph partitioning algorithms to allow for frictionless distributed graph storage and processing are discussed in depth in 3, section 3.2.

Starting with the general technical decision, that horizontal scaling must be favoured over vertical scaling (economic reasons and general state of the art in distributed computing), different distributed computing concepts were studied and evaluated. This, alongside with requirements researchers and software developers would impose on a processing platform, lead to the decision that existing distributed graph processing methodologies can be extended for temporal graph processing. This thesis contributes to distributed graph processing with extensions over Pregel [73] that provide general improvements to the concept but more importantly support the use of temporal concepts in Pregel processes. Together with selected algorithms and query concepts the Pregel extensions are documented in the remainder of 3.

To prove practical feasibility and to allow to methodically test the scalability of the approach a reference implementation of the presented concept was implemented. This reference implementation is based on state of the art distributed computing frameworks and is aimed to allow software developers to implement data analysis projects on top of it. Exemplary applications were implemented in case studies. The software framework is documented in chapter 4 and instructions on how to use the framework together with first results from case studies are documented in chapter 5 [103].

The final contribution of this thesis is the evaluation of the presented framework and its reference implementation. Chapter 6 presents evaluation methods and results from experiments with two temporal graph datasets of different size from the domains of social networks and the web graph [105].

1.8 List of Publications

During the course of writing this dissertation several papers directly linked with the progress of this research were written and accepted at different scientific conferences and journals.

In the first phases of this research the focus was on collecting data from mobile phones such as phone call logs, SMS, e-Mail, and Bluetooth proximity data which can be used to create a temporal network that reflects to a group's social network [109]. Later the overall architecture and a broader outlook on the project was presented. It was discussed how mobile sensor data forming a temporal network can be interpreted to create new application scenarios [108]. Finally details about the distributed temporal graph storage and processing system were discussed as work progressed. Details about the vertex-centric data-model and preliminary tests with established distributed computing

frameworks where discussed in [110] and details on the Pregel-style processing approach where discussed in [111] and [105].

Published Work

Matthias Steinbauer, *Sensor Based, Automated Detection of Behavioural Stereotypes in Informally Formed Workgroups*, Masters Thesis, Johannes Kepler University Linz, 2012. [102]

Matthias Steinbauer and Gabriele Kotsis, *Building an Information System for Reality Mining Based on Communication Traces*, in Proceedings of the 15th International Conference on Network-Based Information Systems, Melbourne, 2012. [109]

Matthias Steinbauer, Ismail Khalil, and Gabriele Kotsis, *Reality Mining at the Convergence of Cloud Computing and Mobile Computing*, ERCIM News, 04-Mar-2013 (invited/not refereed). [108]

Matthias Steinbauer and Gabriele Kotsis, *Platform for General-Purpose Distributed Data-Mining on Large Dynamic Graphs* presented at the 22nd Workshops on Enabling Technologies Infrastructure for Collaborating Enterprises, Hammamet, Tunisia, 2013. [110]

Matthias Steinbauer and Gabriele Kotsis, *Towards Cloud-based Distributed Scaleable Processing over Large-scale Temporal Graphs*, presented at the 23rd Workshops on Enabling Technologies Infrastructure for Collaborating Enterprises, Parma, Italy, 2014. [111]

Matthias Steinbauer and Gabriele Anderst-Kotsis, *Using DynamoGraph: Application Scenarios for Large-scale Temporal Graph Processing*, presented at the 17th International Conference on Information Integration and Web-based Applications & Services, Brussels, Belgium, 2015. [103]

Matthias Steinbauer and Gabriele Anderst-Kotsis, *DynamoGraph: A Distributed System for Large-scale, Temporal Graph Processing, its Implementation and First Observations*, Temporal Web Analytics Workshop at WWW2016, Montréal, Canada, 2016. [104]

Matthias Steinbauer and Gabriele Anderst-Kotsis, *DynamoGraph: Extending the Pregel Paradigm for Large-scale Temporal Graph Processing*, International Journal on Grid and Utility Computing, Volume 7, No. 2, 2016. [105]

Matthias Steinbauer, Markus Hiesmair and Gabriele Anderst-Kotsis, *Making Computers Understand Coalition and Opposition in Parliamentary Democracy*, Lecture Notes on Computer Science, Electronic Government, Volume 9820, Springer, 2016 (received outstanding paper award). [106]

1.9 Outline

The remainder of this work is structured as follows. In chapter 2 related work from relevant fields such as temporal graph algorithms and cloud-based distributed computing is discussed. Chapter 3 the architecture and methods of the distributed temporal graph processing framework are discussed in detail and a reference implementation in Java is discussed in chapter 4. An introduction on the usage of the reference implementation is presented in chapter 5 along with some case studies. The systems evaluation and the results are presented in chapter 6 and chapter 7 closes this thesis with concluding remarks.

Chapter 2

Related Work

This thesis touches many different aspects in the field of computer science such that a structured literature review in the related areas becomes inevitable. This chapter first touches upon the formal concept of temporal graphs (section 2.1) and temporal graph metrics (section 2.1.1), and their possible use in dynamic graph algorithms (section 2.1.2). In contrast to that, more application oriented topics in computer science are used as the foundation layer of this thesis to allow for the proclaimed large-scale temporal graph datasets.

2.1 Temporal Graphs

The concept of a temporal [55] and a dynamic graph [43] have been discussed for many years in literature. A comprehensive summary of the topic is provided in [58]. At first the properties *dynamic* and *temporal* seem interchangeable. However, if the field is closely studied it becomes clear that a dynamic graph describes a graph that in general *can* change over time. The model of a temporal graph is extending over the dynamic graph as it also *tracks* changes in a graph.

According to Harary [43] a dynamic graph is defined in accordance with classic graph definitions where the graph G is a pair (V, E) with V being the finite set of vertices, and E the finite set of edges. With E containing unordered (undirected graph) pairs $\{u, v\}$ of distinct edges. Formally we can call a graph G *vertex-dynamic* if the set V varies over time and *edge-dynamic* if the set E changes over time. Consequently graphs can be both *vertex-* and *edge-dynamic* at the same time.

The foundational work of Harary already states that one prospective way of modelling a dynamic graph is as a sequence of static graphs. So we can say the temporal graph

T is a set of graphs $G_t = (V_t, E_t)$ where any $G_t \in \{G_1, G_2, G_3, \dots, G_n\}$. G_t is a static snapshot of T at time t .

In other related work (Kempe et [55] as an example) use a temporal graph model which is based on timestamped edges. The concept is also known as edge labeling. This is instead of having multiple snapshots of the temporal graph the general definition of the graph is slightly changed to accommodate for the temporal aspect. Then an edge-labeled, temporal graph is a graph L is a pair (V, E) with the same behavior as discussed above but for the set E . Less formally defined as in [55] this means E is an *inconclusive* set of edges. Having each $e \in E$ labeled with a timestamp denoting the point in time the endpoints of e "communicated".

2.1.1 Temporal Graph Metrics

In the context of this thesis temporal network metrics are an important topic in contrast to the static metrics already discussed in the preliminaries (section 1.3). This is also reflected by an increased interest in the topic over the recent years in several different formats. A publication worth highlighting in this context is *Temporal Network Metrics and their Application to Real World Networks* by John Kit Tang [116] which provides a comprehensive introduction to the topic of temporal networks and their application areas, and then discusses different temporal network metrics such as temporal distance metrics, temporal centrality measures, the impact of time on information spreading, and temporal reachability in greater detail.

It is important to note that certain metrics we see as given in the case of a static graph have drastically different meaning in the temporal case. For instance reachability measures and thus paths describe a different concept in both cases. On static graphs the metric of a shortest path is applied in many different application scenarios such as path finding in road networks, and computing distances between actors in social networks which can be used for optimal information dissemination. However, as already extensively argued links in a social network are not necessarily persistent. These links can break such that a computed shortest distance in a graph is not necessarily the optimal path for information dissemination. A shortest path provides the minimum number of hops necessary to reach a vertex b starting at a . Such a path does not contain any information about time. This makes it necessary to define a temporal path and to measure its temporal distance [115]. In contrast to the length of a shortest path (number of hops) the temporal distance gives an estimate on the time it takes to transport information from a vertex a to b .

Tang et al. define a temporal path between the two vertices a and b as follows: $p_{ab}^h(t_{min}, t_{max})$ is the set of paths starting at a and ending at b such that each passes through nodes $n_1^{t_1} \dots n_j^{t_j}$ with the restriction that $t_{i-1} \leq t_i$. Less formally speaking path segments can only continue a path that was already connected in the current timeframe t_i or in any previous time segments. This is in close logic with the message sending idea given before. A message cannot be sent onwards if it was not received at an earlier time. h denotes the *horizon* of the path it refers to the maximum number of temporal hops a path might use. It thus gives an upper bound on the time a message can take to propagate from a to b .

Consequently a temporal distance $d_{ab}^h(t_{min}, t_{max})$ is the measure for the number of temporal hops it takes to move a message from a to b . In accordance with the restrictions given above d is ∞ if no valid temporal path between a and b exists. Similarly other well known metrics have their temporal counterpart defined in literature: Temporal centrality [56], temporal clustering [86], etc. many of which are summarized and formally analyzed in [60]. And other metrics are found only over temporal graphs for instance link delay and triadic closure [126] to name only a few.

In recent related work important aspects of information dissemination in temporal networks are discussed. In a very natural manner a question to ask over a temporal graph is as how fast information can spread in the network. Thus logically temporal graphs can be categorized as *fast* and *slow* networks. A fast network is a network for which all-to-all information spread can be achieved with a high probability in a single time period [4].

2.1.2 Dynamic Graph Algorithms

As touched in the intro of this section a graph can be classified as dynamic or temporal graph. But also algorithms over graphs can fall into these categories. Over a temporal graph certain metrics can be computed in arbitrary time windows such that metrics can be analysed over time. The class of *dynamic* graph algorithms concerns graphs undergoing change. In contrast to the temporal graph algorithm the dynamic algorithm tries to answer the question, given a certain graph, an already computed metric, and a set of changes (vertex and edge insertions and deletions), is it possible to compute the value of the metric after the changes would have been applied. Clearly the idea is to efficiently update the solution of the metric instead of recomputing (which is the prevalent concept in this thesis).

The idea of dynamic graph algorithms was extensively discussed by Eppstein et al. [29]. The cited work provides a broad overview over the topic and gives a very good

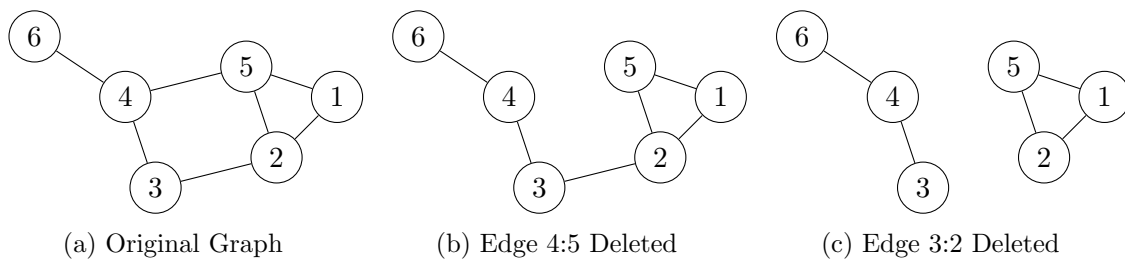


Figure 2.1: Dynamic graph with two edge delete operations

introduction into the related research. In general problem are categorized into classes. A problem is called *fully dynamic* if solution update can be performed regardless of the type of graph updates. In contrast a problem is only *partially dynamic* if only certain types of update operations are allowed. In more detail a problem can further be *incremental* if efficient solution update is only possible on insert operations and *decremental* if this is the case only for deletions.

Clearly there are many cases where the amortized time for updating the solution outperforms the time to recompute the solution. To highlight with an example let us consider a simple undirected graph over which the all pairs shortest paths need to be computed (i.e. as input for a clustering algorithm). Over undirected graphs Dijkstra's algorithm [21] is the most well known solution with a runtime complexity of $O(V^2)$. Optimized solutions using external data structures are providing $O(E + V \log V)$ performance [33] for the same problem.

In figures ??, ??, and ?? a dynamic graph is shown at three stages. In each stage another edge (4:5 and 3:2) being removed. For the graph the set of all pairs shortest paths was computed and recorded in the leftmost table 2.1¹.

Considering the information given now incremental prune and update operations over the result table can be performed instead of recomputing the all pair shortest paths. Let us consider the delete operation which removes the edge 4:5. Clearly all shortest paths containing a path segment [4 : 5] need to be removed from the result set and recomputed. In this case the vertex pairs (1, 4), (1, 6), (4, 5), and (5, 6) require updated solutions (see the updated data in the center of table 2.1).

The second case (removal of edge 2:3) follows the same pattern. A pruning process can remove all affected paths and set out to recompute for the candidates. The interesting thing is, that in this step for none of the pruned shortest paths new paths can be found.

¹In the case multiple candidate paths were available one was chosen at random. And the table does also not contain any reverse paths. If the path from any vertex A to B was already computed then the return path B to A is not in the table as an individual path. These are assumptions and simplifications made to improve readability of the example.

a			b			c		
Source	Target	Path	Source	Target	Path	Source	Target	Path
1	2	{1, 2}	1	2	{1, 2}	1	2	{1, 2}
1	3	{1, 2, 3}	1	3	{1, 2, 3}	1	3	{}
1	4	{1, 5, 4}	1	4	{1, 2, 3, 4}	1	4	{}
1	5	{1, 5}	1	5	{1, 5}	1	5	{1, 5}
1	6	{1, 5, 4, 6}	1	6	{1, 2, 3, 4, 6}	1	6	{}
2	3	{2, 3}	2	3	{2, 3}	2	3	{}
2	4	{2, 3, 4}	2	4	{2, 3, 4}	2	4	{}
2	5	{2, 5}	2	5	{2, 5}	2	5	{2, 5}
2	6	{2, 3, 4, 6}	2	6	{2, 3, 4, 6}	2	6	{}
3	4	{3, 4}	3	4	{3, 4}	3	4	{3, 4}
3	5	{3, 2, 5}	3	5	{3, 2, 5}	3	5	{}
3	6	{3, 4, 6}	3	6	{3, 4, 6}	3	6	{3, 4, 6}
4	5	{4, 5}	4	5	{4, 3, 2, 5}	4	5	{}
4	6	{4, 6}	4	6	{4, 6}	4	6	{4, 6}
5	6	{5, 4, 6}	5	6	{5, 2, 3, 4, 6}	5	6	{}

Table 2.1: All pair shortest paths for the stree stages given in figure 2.1

So it is safe to conclude that the last delete statement has disconnected the graph (table 2.1c).

It has been shown that in certain cases the application of dynamic graph algorithms can provide tremendous speedup. In [40] the concept is implemented for betweenness centrality. For synthetic graphs speedups between 100 and 400 have been reached and for real world collaboration networks they range in 36 to 148.

The system designed in this thesis does currently not make use of dynamic graph algorithms. However, it can serve as a testing ground to evaluate dynamic graph algorithms in large-scale scenarios and to benchmark dynamic implementations against naive implementations over the temporal graph.

2.1.3 Temporal Graphs and Graph Databases

Naturally applied computing translated the models of temporal and dynamic graphs into real world applications. Data models usually serve for in-memory processing but also need persistent counterparts for long-term storage of the data.

Graph databases are used as efficient means of storing, updating, and retrieving graph data. As the enumeration already shows the graph in a graph database is by definition able to change over time. Such that a graph database stores a dynamic graph.

The existing implementations for graph databases such as Neo4j² database are highly efficient and support features one expects from a modern database system such as consistency models and transactions. In general graph data can be well distributed

²Neo4j: <http://neo4j.com/>

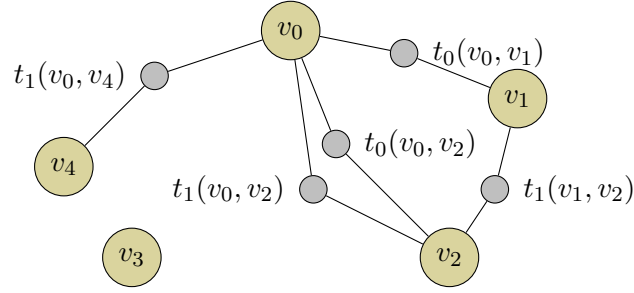


Figure 2.2: A graph with intermediate, temporal vertices between edges

among multiple partitions such that also very large datasets can be stored in graph databases. A famous representative of these distributed graph databases is Titan³.

Graph databases provide extensive query languages and models that allow application developers to run query against the data base and retrieve the results back in their application. Complex algorithms that compute graph metrics cannot be submitted to the database and executed with the concept of data locality in mind.

Moreover graph databases are designed for dynamic and static graphs and thus lack features for the temporal aspects. For both of the presented systems extensions for temporal data storage and querying exist with different implementations.

In Neo4j temporality is emulated through intermediate, typed vertices. These vertices refer to either a point in time or to a timespan and thus allow the application developer to connect vertices with temporal concepts. In figure 2.2 a graph with temporal concepts is represented. To point out two examples the vertices v_0 and v_2 are connected at time t_0 and t_1 whereas v_0 and v_1 are connected at t_0 only. Obviously this way of modeling temporality allows for simple queries of data with time aspects. For instance a query for all vertices labeled t_0 and their adjacent neighbours would allow retrieval of all vertices that have mutual connection at time t_0 . Neo4j provides query language support to retrieve temporal concepts [15]. In the cases where algorithmic processes are executed over the temporal graph this way of graph representation is significantly slower. Oftentimes metrics are computed over in-going and out-going edges which in this case would mean that to determine the connection-weight of two vertices at a certain time-frame would require lookup and processing of multiple edges and intermediate vertices. Which is obviously slower compared to reading edge attributes.

The Titan graph database in contrast allows applications to implement time as attribute in the data. This means that vertices and edges can carry a time attribute which needs to be evaluated during reasoning over the graph. Edges between nodes can show up at

³Titan: <http://thinkaurelius.github.io/titan/>

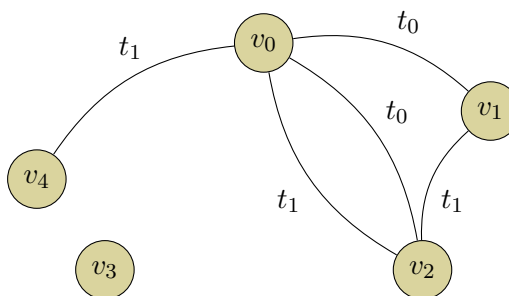


Figure 2.3: A multi-graph with time attributed edges

multiple times thus this storage model requires that the graph model is implemented as a multi-graph. This conforms with the concept of temporal graphs also described in [55].

Figure 2.3 shows a graph with time-attributed edges which reflects the same network and time information as the graph in figure 2.2. It becomes clear that any computation performed over such a multi-graph in worst case requires to read all edges of a vertex in order to determine whether or not an edge for a given time period exists. While in theoretical algorithmic analysis the edge iteration and traversal of an algorithm will in any case (multi-graph or graph) contribute to the worst case runtime complexity with $O(e)$. It is clear that real world implementations will suffer from a far larger number of edges found in the multi-graph.

2.2 Large Graphs

As the problem size grows also the networked data can grow. A graph growing over the obvious limitations of a computer (processing and RAM) requires more sophisticated mechanisms for processing. One of the most obvious ways of mitigating the memory limitation of machines is to move the model to larger background memory on disk. This is where the graph databases discussed in the last section 2.1.3 provide apposite answer but not for distributed and efficient processing.

If computation over networked data is considered different representations are used in applied computing. The prevalent ways of representing a graph in memory are adjacency matrices and vertices with edge lists. For the adjacency matrices naive approaches have atrocious memory complexity of $O(n^2)$. For extremely dense graphs this limitation might not be a problem but for some of the real world graphs discussed in this thesis (social network analysis, technological networks) it is very well known that they form sparse graphs where the naive implementation of an adjacency matrix will result in the detriment of hefty mounts of unused memory. One solution is the use

of sparse matrices as data structure. On the other hand in memory representation of vertices with edge lists also has its disadvantages. Many graph algorithms have efficient formulations as matrix operations which makes their naive implementations over edge list to slow to compete.

For some of the networks presented in this thesis the resource limitations of a single computer are a realistic boundary. For instance the Click dataset used for evaluation (see section 6.4.2) has uncompressed on disk storage of the raw data of just above $13TB$. Independently of the in-memory model chosen, clearly this amount of data cannot fit into memory of state of the art commodity hardware. In general large-scale graphs are a topic of great interest such that multiple datasets of large scale exist.

Many of the datasets available for research are collected in collections. In context of large-scale graphs the most important collection is the Stanford Large Network Dataset Collection [68] (sometimes referred to SnapNets). At the time of writing the collection contained well over 100 distinct datasets from around 15 different backgrounds. Some of the largest graphs found in the collection are online social networks and meme / topic networks. In the first category networks from Orkut⁴ with around 3 million actors and 117 million edges and Friendster⁵ with 65 million users and 1.8 billion edges are the largest.

The Orkut and the Friendster datasets were collected in a study which compared structural communities in online social networks (as those found by a community detection algorithm or perceived by observers) with so called ground truth communities (groups which users of the network actively joined) [123]. For details on the many other networks found in SnapNets the interested reader is referred to the extensive online documentation⁶ of the collection.

Another interesting collection of networked data is The Koblenz Network Collection [61]. It is not focused on large networks but some of the networks listed there are of huge scale. For instance Twitter⁷ crawls with 52 million profiles and around 2 billion edges and data from LiveJournal⁸ with 10 million profiles and 112 million edges can be

⁴The Orkut online social network was acquired by Google and shut down in 2014. The network was very popular in certain areas of the world such that at peak the platform was used by over 300 million users. <http://www.orkut.com>

⁵Friendster was founded in 2002 and in 2004 was the largest online social network worldwide. Later it was outperformed in number of users by the more popular network MySpace. Friendster tried to refocus to remain in the very competitive online social network industry but then shut down its services in 2015. <http://www.friendster.com>

⁶Stanford Large Network Dataset Collection: <http://snap.stanford.edu/data/>

⁷Twitter is an online social networking service with relatively loose coupling between its users. It is famously known for allowing its users to communicate with messages limited to 140 characters, so called "tweets". <http://www.twitter.com/>

⁸LiveJournal is an online social network that allows its users to create blogs and journals online. <http://www.livejournal.com>

found together with an extensive collection of Wikipedia graphs in various languages, categories and sizes.

As large graphs are a field under active study in the following related work in the area of distributed storage and processing of very large graphs is discussed.

2.3 Parallel and Distributed Computing

A natural approach to provide the resources to store and process very large graphs is via scaling. In general computing resources can be scaled vertically as in adding more physical resources to single machines (larger memory and disk capacity, adding processors). The down-side of this approach is that computers quickly hit limitations with this way of scaling (the number of processor sockets on a mother board, the maximum memory size that can be addressed by the memory subsystem, etc.) and in commodity hardware vertical scaling is usually only done in very expensive enterprise mainframe computers. The current prevalent model for scaling thus is horizontal scaling as dividing a problem over multiple computers each of which responsible for a smaller partition.

Current state of the art for distributed computing sees two important trends. There is the classical high-performance computing (HPC) community which is mainly science driven and uses large-scale homogeneous super computing clusters. This is in contrast with the still trending topic of using cloud provisioned computing resources to gain insight into large-scale datasets which is coined as the field of Big Data. Moreover the following paragraphs and sections will reveal that the lines between these two trends dislimn. We see systems using traditional HPC frameworks over cloud computing infrastructure and we see the use of Big Data systems on super computers.

2.3.1 Distributed Matrix Processing

Distributed matrix processing was already discussed in very early days of distributed computing. Homogeneous distributed processors organized as hypercubes were one of the prevalent computing model. Typical tasks included Gaussian elimination with large numbers of variables. This very computing model performs drastically better over dense matrices whereas real world networks often have sparse adjacency matrices [79].

Formally many graph algorithms have efficient solutions over the adjacency matrix of the graph. Especially for iterative stochastic processes over the adjacency matrix,

matrix multiplication is one of the most important operations. For the storage of sparse graphs (as they occur in real world problems such as social networks) sparse matrices can be used as memory efficient data-structures. However, as even these sparse matrices grow to larger sizes with even larger graph models memory of single computers might become insufficient. In [53] a system is presented which is capable of distributing sparse matrices over multiple compute nodes. In order to achieve this the distributed filesystem HDFS, which is part of the popular open source MapReduce [23] implementation Hadoop, is used. Kang et al. [53] as a basic processing mechanism provide distributed sparse matrix processing on top of Hadoop. The operation is named **GIM-V** (Generalized Iterated Matrix-Vector multiplication).

The adjacency matrix of a graph has a big advantage over other matrices in that generally the order of columns and rows can be adjusted under certain restrictions. This can be used to optimize the layout of the matrix or the graph over multiple partitions. Kang et al. show that **GIM-V** allows to implement many popular graph algorithms such as PageRank, the computation of the graph diameter and radius.

The work also provides interesting insight over real-world experiments with a static snapshot of a web-graph obtained from Yahoo. The graph consists of around 6.7 billion edges and is analysed as a small set of connected components making the whole graph a giant connected component. Similar analysis is performed with several snapshots of social network data from LinkedIn for which it is shown that the small world property holds. The diameter stays below 7 in all their experiment runs.

2.3.2 High-Performance Computing Model

In contrast to the adjacency matrix graphs can also be represented as discrete objects and link-lists that hold the edges / arcs between them. In science many large-scale problems have been solved by more general purpose parallel computing models. All around the world data-centers running various versions of MPI are used to distributed all sorts of scientific problems. In [70] challenges that come with parallel graph processing are addressed. As opposed to traditional large-scale problems graphs drive data-driven computation, they are unstructured, have poor locality and have a high data access to computation ratio. Traditional methodologies for scaling fail and problems such as to how design task granularity, whether and how to use global memory, and how to efficiently balance storage and computing workloads need to be addressed. Lumsdaine et al. also briefly touch upon the fact that besides all the complex technical aspects of this topic still software developers require simple programming primitives to work with in order to allow for stable software.

A first group of systems that fall into the category of high-performance computing models are the natural combination of using (graph) databases as a storage backend and a distributed computing approach as a processing frontend. In [80] such a system is described which uses a central router component that through the use of a hashing function is able to route read- and write operations to the correct graph partition. As a storage backend the document store CouchDB⁹ is used. In general the presented work does not provide temporal graph representation and its core focus is not on processing but on the distributed storage of graphs. The system provides extensive mechanisms for graph data replication to provide for better data locality while distributed algorithms run.

A similar approach is presented in [45] using the already discussed Neo4J database as backend storage. In contrast to the focus of this thesis no temporal aspects are addressed and also the sharding mechanism is not used. The presented system makes use of the METIS [54] graph partitioning scheme and runs a local Neo4J instance for each partition. In this respect it is superior to *DynamoGraph* which in currently only provides hashing schemes. One of the main outcomes of this paper is that the METIS scheme provides a clear performance benefit (30% for social network data, 15% for recommendation network data) over naive hashing. As a processing framework GoldenOrb is used which is described in greater detail in the next section 2.3.3.

In contrast to the previous two approaches distributed in-memory systems are used in another line of research. The rationale behind this is that disk-bound systems have a natural bottleneck in a computers IO-performance which causes high performance penalties in graph processing due to the high data access to processing ratio. Through the use of faster RAM this problem can be diminished. The project Trinity [101] is based on a distributed memory cloud. The distributed memory cloud is a in-memory key value store that is organized in so called memory trunks. These memory trunks have disk based backup copies. Obviously in a distributed memory cloud chunks of data cannot be addressed directly by its memory address but Trinity makes use of a hashing mechanism. A key concept in Trinity is the segregated storage of vertices, edges and metadata which keeps the individual storage blocks slim and allows the use of indexing mechanisms. In Trinity multiple processing paradigms are implemented. One is based on Pregel [73] (see details in section 2.3.3) and others are traversal based lookups. The latter can be used for instance to search attributes of neighboring vertices (i.e. a user in a social network looking for friends of friends with similar interests). As a performance indicator it is stated that Trinity can answer 3-hop reachability queries on a 104 billion edges graph in around 100 milliseconds on average.

⁹Apache CouchDB: <http://couchdb.apache.org>

The in-memory processing concept of Trinity finds its extensions in Chronos [42] and later in ImmortalGraph [77]¹⁰. Chronos and ImmortalGraph add methods for temporal graph processing to the in-memory processing concept¹¹. In Chronos the data locality problem is addressed, in general a temporal graph exhibits locality problems in the time- and structure dimension.

The Chronos engine can store static snapshots of a temporal graph and aims to store multiple snapshots of the same vertex close to each other. This behaviour is later exploited in the processing engine where a single algorithm executed over multiple snapshots is computed in so called batches. Batch operations associated with a single vertex are processed in a block such that time consuming operations such as pulling data from other partitions only happen once per vertex instead of once per vertex and snapshot. This comes with the drawback that this exact performance driving behaviour cannot be exploited in situations where data is continuously added to the system. Moreover adding data to the model can and will cause in-memory reorganisation.

ImmortalGraph [77] introduces the snapshot group as a basic unit of storage in the system. A snapshot group describes a timespan in which graph manipulations took place. The temporal graph information can be extended incrementally as new snapshot groups can be added. Apart from that ImmortalGraph provides the same API functions (vertex and edge queries, Pregel based processing) as Chronos.

2.3.3 Cloud Computing and Big Data

Already in the last sections hints towards a processing concept named Pregel were given. Pregel [73] is a highly-scalable, vertex centric, distributed processing concept for graphs. It originates from the trend of processing datasets of growing volume, velocity and variety (3Vs [65] of Big Data [71, 49]). In this arena the embarrassingly parallel processing paradigm MapReduce [23] marks a history of tremendous success.

The general idea of the MapReduce paradigm is that many data processing tasks can be split into a load, *map*, shuffle, *reduce*, and write phase. The load, shuffle, and write phases are mostly task independent and thus have generic implementations in the corresponding MapReduce implementations (e.g. Hadoop¹², MongoDB¹³). Software developers applying MapReduce are left with implementing the map and reduce phases

¹⁰Trinity, Chronos and ImmortalGraph are projects by Microsoft Research

¹¹Looking at the timespan when these projects moved from large-scale (2013) to temporal (2014), and large-scale and temporal (2015) it is clear that *DynamoGraph* addresses a current problem.

¹²Hadoop MapReduce: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

¹³MongoDB MapReduce documentation: <https://docs.mongodb.com/manual/core/map-reduce/>

of processing pipelines. These phases (and also read and write) can be executed fully distributed. This makes MapReduce programs tremendously scaleable.

Pregel is distributed graph processing using the MapReduce paradigm over edge lists of a graph. In its general idea Pregel implementations (for instance Giraph¹⁴, GoldenORB [95]) thus can use generic MapReduce cluster infrastructure for processing.

The success of Big Data projects also lends itself to the great efforts that were made to provide an extensive open source ecosystem. Most famously the Apache Hadoop¹⁵ project, soon after the original MapReduce paper, started to create implementations of the Hadoop Distributed File System which closely resembled the Google Distributed File System and a MapReduce implementation running on top of it. This was the starting point of many other open projects that used this infrastructure as their backbone: HBase¹⁶ (BigTable implementation [16]), Hive, etc. In Apache Giraph¹⁷ an open source implementation of Pregel for the Hadoop ecosystem can be found.

In general it has been found that Pregel is suitable in application scenarios like the ones presented in this thesis. Especially social network analysis algorithms such as computing clustering coefficients, finding components (connected components), estimating the diameter and degree of separation computation, have efficient implementations in Pregel [93]. However, in the real world implementation of Pregel algorithms we see a reoccurring pattern where algorithms have to break out of the massive parallel processing paradigm. Where a Pregel job is designed to theoretically run a small function in parallel for all vertices in a graph still many of the discussed algorithms require aggregation. This is why extensions over Pregel have been proposed and tested which allow for simpler and more efficient algorithm formulation. These extensions are the possibility to run multiple jobs in a sequence, to allow global operations for a job to enable aggregation function, and also dynamic repartitioning schemes are proposed [96].

In recent related work we see a change in the Big Data landscape which causes a shift of paradigms. MapReduce jobs have a performance bottleneck which is their disk-bound read and write phase. This together with an upcoming general requirement to process data in real-time lead to the development of newer Big Data processing approaches that allow for massive distributed stream processing (see Apache S4 (incubating)¹⁸, Apache Storm¹⁹) and puts more focus on in-memory processing.

¹⁴Apache Giraph: <http://giraph.apache.org>

¹⁵Apache Hadoop: <https://hadoop.apache.org/>

¹⁶Apache HBase: <https://hbase.apache.org>

¹⁷Apache Giraph: <https://giraph.apache.org/>

¹⁸Apache S4, incubating project; the time of writing this not under active development:
<http://incubator.apache.org/s4/>

¹⁹Apache Storm: <http://storm.apache.org>

The current iteration of Big Data platforms has a prevalence for the Apache Spark framework²⁰. That itself integrates well with the existing Hadoop ecosystem. In general Spark can use data that is living on Hadoop driven stacks such as off of the Hadoop distributed file-system and Hadoop based databases such as HBase, but it can also run standalone. Most importantly it provides the current state of the art Big Data processing framework. Developers are liberated from creating MapReduce jobs to describing data-processing pipelines which are automatically translated into optimal processing paradigms.

This new processing paradigm is also the foundational layer for a new graph-parallel framework on top of Spark which is called GraphX [122]. GraphX extends on Spark's Resilient Distributed Datasets (RDD) and provides a graph datastructure called the Resilient Distributed Graph (RDG). These are tabular representations of vertices and edges and provide extended functionality such that data in the RDG can change without affecting running algorithms. Thus enabling processing of *dynamic* graphs.

Still the related practical systems up to now do not provide direct interfacing of *temporal* graphs. In very recent related work Moffitt and Stoyanovich present a data-warehouse approach for temporal graphs called Portal [78]. As opposed to *DynamoGraph* which is discussed in this thesis Portal provides a declarative query language which extends over SQL to allow efficient querying and exploratory data analysis on temporal graphs. The approach lends itself very well to practitioners in data-warehousing and analytics since reoccurring patterns in their model are queries over RDD's which represent vertices and edge-lists. In general it is shown that more general algorithms (such as PageRank [89]) can be implemented as language extensions.

2.4 *DynamoGraph* Distinctive Features

The work presented in this thesis results in a software prototype called *DynamoGraph*. It is a distributed processing framework which in the exact same realization is yet to be found in related work. The generic storage backends provided by graph databases either do not fulfill the requirements to provide efficient distributed computation over the stored data or do not provide out of the box support for temporal graph structure and reasoning.

In the high-performance computing arena the processing approaches often require custom machines and do not support the state of the art cloud computing models where distributed processing resources are provisioned on demand. This is possible with mod-

²⁰Apache Spark: <https://spark.apache.org>

ern Big Data systems which are designed to provide scalable storage and computing resources but no self-contained solution for reasoning on large and *temporal* graphs is available.

Recent related work [78] shows that the demand for large-scale, temporal graph processing exists. It is safe to assume that systems like Portal are in a similar stage of development as *DynamoGraph* but have a different focus.

Chapter 3

Distributed Temporal Graph Processing Framework

As presented in previous chapters temporal graphs are a natural choice for modelling social network systems. Since these networks tend to grow very large in size and also their historical changes need to be tracked to create generate new findings on top of social networks the size dimension is even more problematic. In this chapter a system for distributed temporal graph processing is presented. The distributed storage and processing system is designed in a way that it supports horizontal scaling and thus can adapt to growing temporal graphs. This system is implemented as a generic graph processing system such that it is also applicable to other dynamic graph processes such as protein to protein interaction, and changes in internet network behaviour.

3.1 Temporal Maps as Data Structure

The framework presented in this thesis operates on temporal graph data. In general there are two widely used forms of representing graph models in memory: adjacency matrices (as described in related work 2) and vertices with link lists. As already highlighted in related work and preliminaries memory usage for graph storage reaches $O(n^2)$ worst case complexity (very dense graphs) such that in-memory representation for very large scale graphs will exceed memory sizes available on single computers. Which raises the demand for distributed computing approaches (details on the distributed computing models are discussed in section 3.3). For adjacency matrices distributed matrix processing frameworks can be used [53] which allow to implement matrix based graph algorithms in distributed computing environments.

The distributed computing framework explained here uses vertices with linked lists that hold the pointers (arcs, edges) to other vertices are used to store the graph data. This

is done mainly for the following reasons: In distributed computing environments it is just easier to store vertices together with link information as independent documents, because this way it is assured that all link information is available on the local machine when accessing a vertex. Further these documents can integrate very well with modern Key-Value and document storage systems. And finally the temporal aspect of a graph can be modeled in a more elegant way than on top of matrices.

In the presented system vertices are described in a document structure that itself is a map of key-value pairs called the *profile*. In general arbitrary information can be stored in maps, especially sets, lists and again maps can be stored as the value of any key-value pair. Certain keys of the map describing a single vertex have special meaning to the framework. These keys are an *identifier* which uniquely identifies a vertex in the system and *edge-lists* for the incoming and outgoing edges of a vertex.

An example profile from a social network analysis task can be found in listing 3.1. The document is formatted in JSON [1] which is widely used in social networking and semantic web applications. The profile *39827736* describes a person named *Rob Henderson* and is connected via one outgoing and one incoming arc to the vertex *39761932*.

Listing 3.1: Example vertex profile in JSON notation

```

1 {
2   id: 39827736,
3   name: 'Rob Henderson',
4   description: '',
5   inEdges: [ {
6     weight: 7.3,
7     edgeType: 'PHONE',
8     source: 39761932,
9     target: 39827736, } ],
10  outEdges: [ {
11    weight: 10.0,
12    edgeType: 'EMAIL',
13    source: 39827736,
14    target: 39761932, } ],
15 }
```

Since the model contains a temporal dimension also the basic data-structures used to describe this model need contain a temporal component. The design of the presented framework is based on maps capable of storing a temporal dimension. A temporal map data-structure is a map that is sliced up in time-frames of a certain size. The size of

the time-frame is called the maps *resolution*. Data elements (key-value pairs) stored in the temporal map are always living in a certain time-frame. Thus a temporal map is a list of regular maps; one map for each time-frame in the temporal map.

Insert operations on any temporal map always need to specify a point in time. The temporal map determines the time-frame the data element needs to be inserted to, and inserts the element in the map associated with the time-frame the specified time lies within. The insert operation i on any temporal map M can be defined as $i(M, k, v, t)$ where k denotes the map key to be addressed, v refers to any arbitrary data to be inserted and finally t describes the insertion time-stamp.

Read operations in contrast need to specify a time-frame in which the read operation is to be executed. The read operation queries all regular maps for the specified time-frame to retrieve data elements and returns the data from these maps found by querying with the specified key. Formally a read operation r over any temporal map M can be defined as $r(M, k, b, e)$. Where k denotes the key to be retrieved, b marks the begin and e marks the end of the time-frame to be considered by the read operation.

Obviously this behaviour can and will lead to conflicts where a data element to a certain key occurs in different time-frames with different values. Such that corrective measures need to be taken in order to resolve occurring conflicts. Several different strategies can be used to resolve conflicts in temporal maps.

Temporal preference uses the temporal information stored in the map to retrieve the youngest or the oldest element returned for the key k . The retrieval of the youngest element is default behavior.

Temporal aggregate: In situations where the evolution of a certain attribute in the temporal map is being addressed temporal aggregate can be used as conflict solving strategy. This is actually not a conflict solution but just aggregates all results found for k in a new map where the time-stamps assigned in the temporal map are used as key.

Aggregate functions can be used where users are only interested in mathematical aggregates of values stored in the temporal map. The used aggregate function such as *maximum*, *minimum*, *average*, and *median* can be specified as function argument and is computed during the read operation in the temporal map.

Set aggregation: In the case the data type of values stored in the temporal map is of set, or map then a special set aggregation strategy can be used. This will combine

all values found for a certain k into a new collection containing all the elements found in the temporal map.

The listing 3.2 shows the same vertex profile already presented in listing 3.1 converted to a temporal map. All the maps key-value pairs are now wrapped in maps identified by time-stamp keys. A new optional property *resolution* denotes that the temporal map is being stored with a month based resolution.

The example shows how the month-long time-frames generated by the month resolutions setting are referred to by the UNIX-time-stamp describing the begin of the time-frame. In the example profile the first time-frame is indexed with *1420070400* which is the 1st of January 2015 and thus refers to the month of January 2015.

Listing 3.2: Example temporal vertex profile in JSON notation

```

1 {
2   id: 39827736,
3   resolution: 'MONTHS',
4   '1420070400': {
5     name: 'Rob Henderson',
6     description: '',
7     inEdges: [ {
8       weight: 3.3,
9       edgeType: 'PHONE',
10      source: 39761932,
11      target: 39827736, } ],
12    outEdges: [ {
13      weight: 4.0,
14      edgeType: 'EMAIL',
15      source: 39827736,
16      target: 39761932, } ],
17  },
18  '1422748800': {
19    inEdges: [ {
20      weight: 4.0,
21      edgeType: 'PHONE',
22      source: 39761932,
23      target: 39827736, } ],
24    outEdges: [ {
25      weight: 6.0,
26      edgeType: 'EMAIL',
27      source: 39827736,
28      target: 39761932, } ],

```

```

29     }
30 }

```

Typical queries against this temporal vertex profile might be one of the following: "Retrieve the current *name* of profile 39827736?" This question can be answered by applying *temporal preference*. This can be executed by finding the *name* property starting in the youngest time-frame. Search stops when the most recent *name* property was found.

A temporal graph G_t instance in this model is defined by a set of temporal maps one for each vertex in the graph such that $G_t = \{M_1, M_2, M_3, \dots, M_n\}$ and n denotes the number of vertices in the graph.

On top of this definition of a temporal graph queries concerning more than a single vertex can be executed. "What is the edge-weight for e-Mail from profile 39827736 to 39761932 in the year 2015?" This query can be answered by applying *set aggregation* on the property *outEdges* on the temporal maps describing the given profile ids which will result in an aggregated set of all *outEdges* found in the addressed profiles.

3.2 Graph Partitioning Strategies

The presented framework aims to handle temporal graphs that grow very large in size such that it is not feasible to handle data and processing on a single computer. This means that the temporal graph needs to be distributed on multiple compute nodes. A large-scale temporal graph in general can be split along two dimensions. Firstly it can be split along the temporal dimension. This is especially useful if data is to be analysed in predictable time-frames and if older data become obsolete over time. Secondly a split in a temporal graph can be structural as grouping certain vertices of the graph together in a partition. This form of splitting is useful if possible application scenarios require analysis of non-predictable time-frames on the whole time period under observation. In this work structural partitioning was chosen over temporal partitioning since (1) proven algorithms exist for graphs partitioned in such a way, and (2) because the analysis time-frames in the important application domain of social network analysis are hard to predict.

The problem of how to split a given graph into a number of partitions is called graph partitioning in graph theory. In chapter 2 related work to this problem is described in greater detail. In graph partitioning the task at hand is to find optimal splits in a graph where the optimization goal most often refers to minimizing the number of cross-

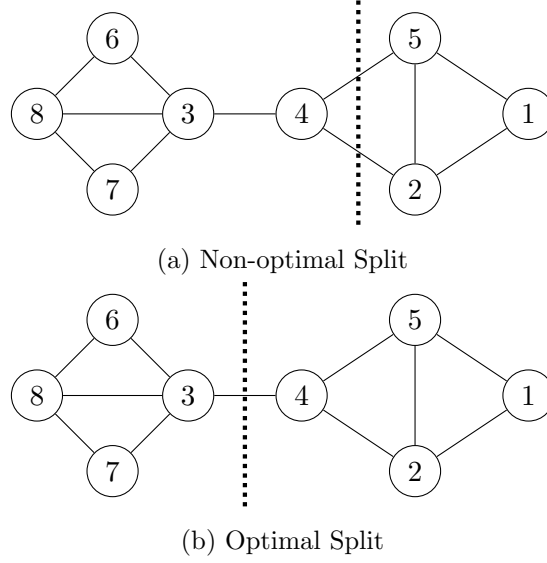


Figure 3.1: Splits in Structural Graph Partitioning

partition arcs while still maintaining similar sized partitions. This is done under the assumption that in many graph algorithms often neighbouring vertices are used during computation. Thus a high number of cross-partition arcs can have severe penalty on the execution time of such algorithms.

Figure 3.1 shows a comparison of two different splits in a graph. In the first case a non-optimal split is shown. The found split cuts through two arcs whereas in the optimal split only one cross-partition arc is needed. Further the non-optimal version is also imbalanced in terms of partition size.

Finding optimal partitions is not in the focus of this work. The framework discussed in this thesis uses partitioning themes from other authors and applies them when distributing a temporal graph over a given number of computers. This is done by computing the *partitioning function* for each newly created vertex $v_x \in G_t$ as in $P = p(G_t, v_x, m)$. The parameter m refers to the partition *priority* which needs to be evaluated by the partitioning function to determine if the computed partition number will be used as the primary/master partition for the vertex to be stored, or will be used for any of the backup partitions where a copy of the vertex is stored for fail-over. The partitioning function can be re-evaluated at any time such that in cases where the number of partition changes the distribution of vertices can be rebalanced.

In a distributed computing environment an optimal function p is one that can be implemented without global knowledge of the complete temporal graph. In a perfect scenario the parameter G_t can be omitted altogether. Computing the partition number for any given vertex only knowing the vertex profile v_x itself can be achieved through three strategies:

- Computing functions over vertex local information,
- lookup-maps that are compact enough to be distributed efficiently in the compute cluster, and
- partitions to be filled sequentially during graph loading.

In the case of functions the partition assignment is computed through a function from properties found in the vertex profile. One such popular function would be to compute modulo over the vertex identifier. Under the assumption that vertex ids are assigned in a linear manner this will lead to a situation where vertices are evenly distributed over the available partitions. This comes with the cost of creating non-optimal splits and thus loosing performance during algorithm execution.

Other functions could analyse whichever attributes of the vertex profile. In a social networking scenario the age of the users, their current location, and similar attributes could be used to create partitions based on attribute clustering. Depending on the attribute chosen for clustering sometimes semi-optimal splits will be found.

To get better partitioning schemes aligned with the task at hand (i.e. clustered schemes in social network applications) fast and distributable lookup maps can be used. These lookup maps are generated and updated by a global algorithm in a defined time interval. The resulting lookup table can be used to determine the partition number for any given vertex profile in the system.

However, in this scenario assigning partition numbers to newly added vertex profiles becomes more difficult. In the proposed framework it would be possible to just use a static partition number for new profiles i.e. just adding all new vertices to partition 0 and waiting for the next update run to re-locate improperly located vertices to a fitting partition.

On the other hand it is also possible to introduce an additional *oracle* function o which creates a good guess on where to place the new vertex. Assuming that for a newly added vertex arcs connecting it to the rest of the network are already available, a good guess will be to assign the new vertex to the partition which it is strongest connected to.

A final strategy for graph partitioning that can be used is to fill all partitions sequentially. Let us assume that for each partition living on a certain computer there is an optimal maximum of vertices that can be handled efficiently on a single partition, and let us further assume that all new vertices added to the system get their vertex

identifier assigned in a sequential manner. If these conditions are met, a partitioning strategy can just sequentially fill every partition to the optimal maximum. In this case the partitioning function P can be implemented based on the delimiting values (min, max) of the vertex identifiers for each partition.

For the aforementioned partitioning functions P it is important that they can be evaluated in vertex local context with and hopefully minimizing the time to compute it. As the following sections will unfold the function P will be used regularly during computation in the framework and thus may impose a huge performance penalty on running algorithms.

Further, partitioning strategies that often require re-organization of the partitions can and will also have bad impact on the overall systems performance. Not only can re-organization cause high workload (computing, network, and data-access) on a distributed system but as per the current design during re-organization all other algorithm execution is being stalled.

One advantages that comes with representing vertices in temporal maps as described in the previous section is that vertices are self-contained. The document or map describing a vertex contains all attributes of the vertex itself but also contains all link information that connects the vertex to other elements in the network. Thus the vertices are mobile in those cases where migration of vertices from one to another partition is absolutely required. This is the case when computing resources and thus partitions are added to the underlying computing system (up-scaling) or when partitions are removed from the system (down-scaling or node-failure). If during a re-organization run vertices need to migrate from one to another partition it is sufficient to move the map describing the vertex to the new target partition. In respect of the properties a partitioning function should have, re-organization runs can be performed locally for each partition. When instructed to reorganize the partitions are inspecting all their locally stored vertices with respect to the partitioning function P . For all vertices where P computes a partition id which requires to migrate the vertex in question the old partition can instruct the new partition to store a copy of the vertex. The vertex can be deleted from the old partition once the new partition confirms successful migration.

The current state of the presented prototype supports the integration of more complex partitioning schemes. As of now only the modulo scheme is implemented. For very large scale graphs modulo partitioning is also very appealing since exact and optimal solutions to graph partitioning show very bad worst-case runtime behavior. As already discussed in the preliminaries (see 1.3.3.2) exact solutions based on the Girvan-Newman [85] approach require prior computation of betweenness centrality which as of now is

computed fastest with the algorithm of Brandes in $O(nm + n^2 \log n)$ [14] which clearly does not scale for very large graphs (large n).

3.3 Parallel Computing Models

It is a clear goal of this thesis to show that efficient processing over large scale temporal graphs is possible by implementing algorithms on top of distributed computing models. This means that for the temporal maps described in section 3.1 which can be scattered over many computers as per the partitioning strategies discussed in 3.2, efficient parallel and distributed computing models need to be found. The following will reflect on basic parallel and distributed computing models and their application on graph and temporal graph data.

Traditionally problems in the domain of graph data and moreover in the domain of temporal graph data have been addressed using parallel algorithms. Observing the arena of graph algorithms one sees many algorithms where even the most naive implementations of basic algorithms such as centrality measures and path following algorithms can easily be run in parallel. They can be implemented to not pose any form of referential data dependency such that issues with locking can be reduced to a minimum. General distinctions between the different parallel computing models can be made regarding a global memory being available to all processing units further referred to as a shared memory model, or memory and processing units being distributed into several independent machines. In the latter case some form of communication network is necessary to allow for data exchange between the processing units.

3.3.1 Shared-Memory Model

In a shared-memory model all processing units (processors) have access to the same shared memory. Such that every processor can read and write any memory block of the complete dataset. This architecture is found in many modern multi-core computers. Since memory is typically a limited resource in a computer blocks of data can be written to hard drives such that a virtual memory exceeding the size of the real installed memory can be provided to applications. Still it is required that the whole model of the problem domain to fit into this virtual memory.

Graph processing algorithms running on shared-memory model often use adjacency-matrix representation for the graph data. There exist memory organisation models

capable of storing sparse matrices in memory which allow sparse graphs to be stored with a vastly better memory footprint than $O(n^2)$.

One of the main difficulties with shared-memory models (not limited to graph processing) is that there also need to be proper locking mechanisms in place that ensure write operations can only be executed by a single stream of operation at any time. This locking obviously diminishes some of the benefits of parallel computing; processors waiting on locks are stalling.

3.3.2 Parallel Message-Passing Model

In order to overcome the limitations of shared-memory models (limited memory, limited number of processors per computer) so called shared-nothing infrastructures can be used. In a shared-nothing system individual computers each equipped with a certain number of processors and local memory are interconnected via a communication network. The communication network allows the individual computers to exchange messages during the course of algorithm execution.

The most important standard with several implementations is the Message Passing Interface (MPI) [41]. In MPI a usually fixed number of computers are connected through a communication network each computer executing the same program. MPI programs usually consist of a data distribution, computation and tear-down phase. At first the data in question is uploaded and distributed in the compute cluster. Then the individual computers perform computation on their local portion of the dataset while mutually exchanging messages that transport results, and algorithm execution information. Exchange of messages is also used for synchronisation, this is obviously not removing the general problem of processors stalling in wait cycles, however locking becomes an implicit function of the algorithm.

Traditionally MPI infrastructures are designed for high-performance computing workloads such that defined sizes of compute clusters are used. This is why the interfaces themselves are designed for static setups where the number of compute nodes participating in computation is fixed. Newer versions of MPI allow for dynamic scaling of the compute cluster, however the software developers need to address changes in the size of the compute cluster in their algorithms to benefit from scaling up clusters. If compute nodes fail intermediate results from the node are lost which oftentimes means that the algorithm has to be started from scratch.

3.3.3 Dynamic Message-Passing Model

In modern computing environments oftentimes cloud computing forms the basis layer of data-centers. This means that the exact communication structure is unknown to the resource consumer. Network topology, workload, and VM load balancing can and will change over time such that a dynamic message-passing model that is capable of adapting to current workload is desirable. Such dynamic message passing models are available in the form of modern Big Data processing frameworks. Currently Big Data batch processing based on the MapReduce [23] paradigm and Big Data stream processing frameworks such as Storm ¹ and S4 ² are state-of-the-art.

MapReduce frameworks allow for compute and storage nodes being added to and removed from the system at any time. During processing phases MapReduce algorithms iterate through the defined phases of read (reading input files), map (mapping input data to key value pairs), shuffle (message passing phase that aggregates all key value pairs with the same key on a single compute node), reduce (aggregates key value pairs according to their key), and write (writing results back to disk). MapReduce systems write all their intermediate results to distributed disk storage such that in the case of node failure (or just a node being removed from the system on purpose) a checkpoint state of all computation is available and the system can continue to execute with only minor delays.

Stream processing frameworks however are closer to frameworks implementing the MPI standard. A number of computing elements performs local computation and is able to communicate with other computing elements for data exchange and synchronisation through the use of messages. Stream processing frameworks however are dynamically scalable in the sense that the exact topology of roles to be assigned to individual computers and processors is delegated to the computing framework. Developers implement their algorithm in writing the local computation for individual processing elements and describing how data is to be sent from one processing element to another. This description is often referred to as the topology. In contrast to MPI based systems the concrete execution of the topology is defined by the framework on runtime. Parameters such as load, network latency, and rules defined in the topology are used to determine an execution plan which is also re-evaluated on regular basis.

¹Storm: <http://storm-project.net>

²Apache S4: <http://incubator.apache.org/s4/>

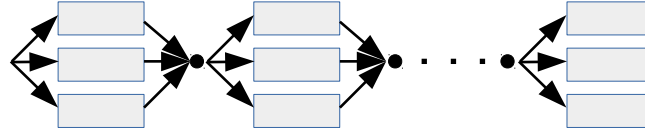


Figure 3.2: Schematic of Compute Aggregate Broadcast Computation

3.3.4 Compute Aggregate Broadcast Computing

One option to reach a truly dynamic message-passing model where the multi-computer is able to automatically adapt to current workloads is by applying a compute aggregate broadcast (CAB) regime [63]. This regime results in a clear partitioning of the computational workload to be performed by an algorithm. Compute phases refer to local computation, aggregation to the collection of results from individual computers and aggregating them into a global result, and finally broadcasting new data and tasks to all nodes of the multi-computer.

The clear structure inherent in this model leads to the following advantages. First of all synchronisation is implicit which means that developers writing payload algorithms in a CAB model can focus on the actual algorithm and synchronisation can be handled by the framework. Further the clear phases of this computing paradigm leads to easy traceability of the overall system's status by the managing framework. When a global system state is available maintenance operations such as resizing the compute cluster, reacting to failed compute nodes, and monitoring operations can easily be introduced.

The computation framework presented in the following paragraphs follows the CAB paradigm. Computation is split into local computation that is executed in the context of a single vertex of a graph. An aggregate phase which aggregates intermediate results stored by individual computers in local intermediate result maps, and a broadcast phase which broadcasts aggregated intermediate results to all nodes and then either instructs all nodes to continue with the next iteration of computation, or in the case the algorithm run has reached its defined end instructs all nodes to halt the algorithm.

A computational model that follows the CAB regime is the bulk synchronous processing (BSP) model which was first described in detail in [118]. A BSP system operates by performing a sequence of *supersteps* which are processed in parallel in a distributed computer. The distributed computer can be imagined as a virtual machine with a set of processors. Each of which is equipped with private memory. The processors can interact with each other using a communication network. In theory the processors are mutually connected with point-to-point communication channels. Algorithms in BSP strictly follow the CAB regime as illustrated in figure 3.2. Parallel *local computation* phases are

followed by *global communication* and a *barrier synchronisation* before further iterations of supersteps are scheduled.

To implement message passing in theory each processor has an outgoing and an incoming message pool. During local computation the processor can add messages to the outgoing message pool and can consume messages from its incoming message pool. Practical implementations (such as the one discussed in this thesis) use a communication medium that operates in parallel to the local computation phases such that the actual execution time for the global communication phase can be reduced to a minimum.

The model through the barrier synchronisation guarantees that any *superstep* $t + 1$ can only be executed after all tasks of the preceding superstep t have completed. This is in contrast to the LogP [22] distributed programming model. LogP starts of with the same basic model of a distributed computer with a set of sequential processors and a communication network. In contrast to BSP there is no explicit synchronisation mechanism. Processors in this model can either be in an *operational* or *stalling* state. Operational processors can perform local computation or submit messages to other processors through a communication channel. Outgoing messages are delivered through the communication network as quickly as possible and stored at the destination processors incoming message buffer. A processor is in stalling state while outgoing messages are in transit and continue in operational state once delivery is confirmed. In conclusion the LogP model is an asynchronous model whereas BSP is an explicitly synchronous computing model.

In formal analytics it was already shown that a LogP model can be simulated through BSP and vice-versa. Such that in any given distributed computation scenario either model can be chosen. Factors such as the general performance and simplicity of algorithm formalisation need to be taken into account when selecting a model. In mutual simulation BSP and LogP did not show any significant formal performance advantages or disadvantages such that the general simpler algorithm formalisation in BSP gives advantage for this model [12].

3.4 Distributed Temporal Graph Processing

In figure 3.3 the basic building blocks of the proposed computing framework and their interplay are shown. Implementations of the framework run on top of distributed compute clusters which are built from compute nodes that are either acting in the role of a master node or in the role of a worker node. The master node is providing an

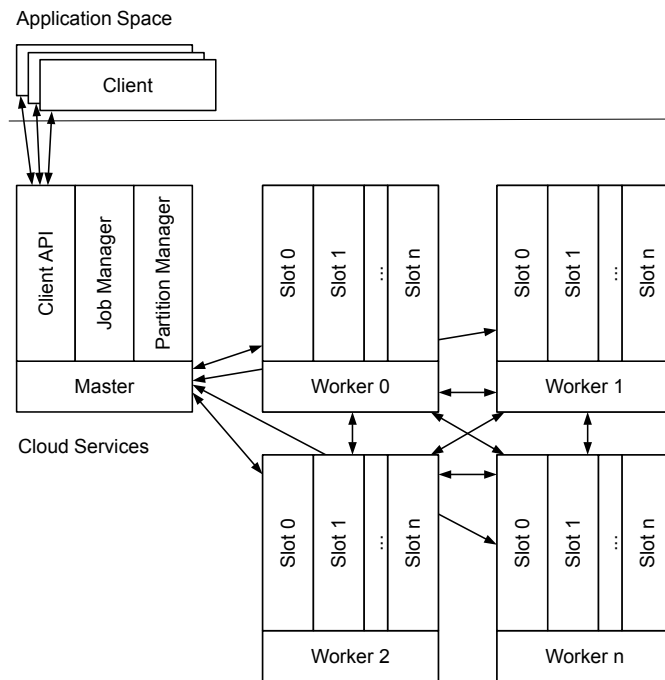


Figure 3.3: Basic building blocks and their interplay

interface to prospective clients of the framework and constantly monitors the whole compute cluster for possible worker node failures. The master node also keeps track of all partitions in the cluster and the worker nodes handling these partitions. Further, the master node keeps a list of all running jobs in the cluster and stores part of their intermediate results, specifically the intermediate state of the global result. The master is capable of assigning new jobs to worker nodes. These jobs are then executed in the compute, aggregate, broadcast paradigm [63] which leads to implicit synchronization steps during the aggregate phase. The master has further control over jobs such that they can be paused after each synchronization step, and canceled completely if an error occurred or if requested by a client.

Worker nodes on the other hand are responsible for hosting graph data and executing jobs on this data. This means that each worker node is responsible for one or more graph partitions and executes compute jobs on its local partitions as assigned from the master. Since modern computers are usually equipped with multiple processors worker nodes are capable of running multiple processing threads in parallel, these processing threads are called slots in the architecture. One slot refers to one graph partition and can strictly process in the scope of the assigned partition only.

The data model used is vertex centric as opposed to a matrix based model often used in other work [53]. The vertex centric model is used because it allows for easier distribution of graph data over many partitions and as discussed earlier in section 3.2 also for easier vertex migration during cluster reorganisation. In this model the arcs are stored as

part of the vertex profile and thus each arc needs to be stored twice, once at the originating vertex and once at the destination vertex. Since the system is operating on temporal graphs each data element in a vertex profile is a temporal element that has a timestamp marking the begin of its existence. Consequently this is also true for arcs and edges. Framework-users are able to specify the resolution of timestamps in the system in order to adapt analysis granularity on the time scale according to an applications requirements i.e. in some cases data condensed on daily basis might be sufficient whereas in other application data needs to be available on a hourly basis.

A single cluster instance can host many temporal graphs at once. Each instance of a graph is uniquely identified by a label which is called its *namespace*. This allows for applications to be implemented where the result of an algorithm i.e. graph clustering algorithms, is again a graph and the resulting graph is even capable of linking back to the original graph such that the relation between original data and computed data can be represented in the system. Again in the case of graph clustering algorithms this can be especially useful for algorithms that compute a hierarchy of clusters [25, 64, 19, 85].

All components of the system are using a communication protocol to directly exchange control and data messages. In general there are three types of messages:

- direct master to worker or worker to master control messages such as messages that regularly check the clusters health status and messages that distribute the current partitioning scheme in the cluster as explained in section 3.2,
- partition specific control messages that are either broadcast to all slots or automatically routed to a certain slot such as the *add-arc* command, and
- vertex to vertex messages that are used during job execution to exchange data between vertices as explained in greater detail in section 3.4.1.

3.4.1 Pregel-style Job Execution

In vertex centric graph systems as in the present work also vertex centric processing paradigms have proven to be successful and moreover very scalable. One of these systems is Pregel [73] which is a system for large scale graph processing implemented at Google. It follows the bulk synchronous processing paradigm as discussed in section 3.3.4 Pregel is in the class of graph parallel distributed computing frameworks. The underlying assumption is that there exists a *compute* function $c(v, \Gamma)$ which is executed in parallel for all vertices. Parameter v refers to the vertex the function is to be executed on and Γ denotes a local copy of global memory that contains i.e. configuration variables

and global results. As visible from the functions signature c needs to be implemented in a way such that the *compute* function is able to access vertex local data only. This means that data access to other vertices needs to be implemented through message passing mechanisms. In Pregel-style frameworks there exists a function $m(v_t, x)$ which can be used to send any arbitrary message x to any other vertex in the system. Computation in Pregel-style frameworks is modelled after a compute aggregate broadcast regime. This leads to systems that perform their computation in steps, these steps are called *superstep* in Pregel. As the system global variable t that progresses through supersteps a Pregel-style computing framework is able to guarantee that vertex messages generated and sent in time-step t are to be delivered to its recipient exactly at time-step $t + 1$.

Also inherent to Pregel-style computing frameworks is a global halting mechanism which is used by algorithms to define a algorithm-halt behavior. Since neither the master nor any of the worker nodes have a global and complete view of the graph and the execution states, also the decision as of when to halt algorithm execution needs to be a distributed decision. The halt mechanism is implemented by a voting scheme where each and every vertex can vote for halt after executing function c . Technically the vote-to-halt can be implemented as the return value of c . Pregel-style frameworks will continue to execute until all vertices have voted to halt the algorithm and there are no more vertex-to-vertex messages to be delivered.

Distributed computing systems implemented in the way described above can be classified as multiple-instruction, multiple-data (MIMD) systems according to Flynn [32].

3.4.2 Extensions over Pregel

The presented framework is inspired by Pregel in many aspects but the concept was extended for functions to perform a global computation after each step (*global*), to perform coordinated initialisation *init*, and mechanisms to handle temporal graphs i.e. processing can be performed in sliding time windows which allows for dynamic properties of graphs to be calculated. These extensions are implemented in the following way.

Global computation is implemented by executing the function $\gamma(\Gamma)$ on the master node between each superstep. This global computation step can be used to aggregate intermediate results and avoids work-around solutions where the function γ needs to be implemented as part of location computation in c . For initialisation another function $\kappa(\Gamma)$ is executed before anything else also in the global context on the master node. Function κ can be used to initialize the algorithm with configuration parameters.

For local computation the function c already known from Pregel is extended by a new parameter θ which is used to describe the time-frame the current algorithm execution run is being bound to. The method signature for temporal Pregel thus looks like this: $c(v, \Gamma, \theta)$. The parameter θ is used inside of c to properly address data from the corresponding time-frame using the mechanisms provided by the underlying temporal map.

Further the framework is also different in the aspect that it does not allow for graph mutations during algorithm execution, which is a valid restriction in many application areas of temporal graphs such as social network analysis where one can assume that information from the past is rarely ever changed.

Compared to the original Pregel implementation the framework presented here also provides mechanisms to store data in the distributed cluster and allows developers to use and implement many different partitioning algorithms. Graph partitioning is not part of the original Pregel specification, the assumption is that graph partitioning was done prior data analysis and data is already stored in a partitioned fashion on top of a distributed file-system.

The Pregel-style computation framework discussed here extends the concept by a multiple-phase algorithm execution concept. Oftentimes graph algorithms are built from multiple building blocks. See for instance the computation of maximum flow [100] which consists of a forward and a backward phase or the famous Clauset, Newman, Moore large-scale graph clustering mechanism [19] which is based on vertex-centrality and arc-betweenness requires several steps to be executed before the actual clustering can be done. A rough execution plan for this clustering algorithm will have the following phases: (1) computation of vertex-centrality for every vertex (trivial), (2) computing arc-betweenness for every arc through counting all shortest-paths that run through every arc, (3) iteratively remove highest ranked arcs until clusters which strong central vertices occur.

Finally, since the computation framework discussed here is closely interwoven with an underlying concept for temporal map storage and graph distribution also changes in data can be used as a mechanism to automatically perform computation. Especially in the case of temporal graphs it makes sense to compute certain metrics in certain time-intervals or whenever data in a certain time-frame changes. The framework provides such mechanisms in the form of triggered algorithm execution.

In figure 3.4 a state machine for the possible global system states are illustrated. Algorithms submitted to the system are in the *started* state and will move to the *init* state which executes initialization routines (function κ at the master. For instance this can

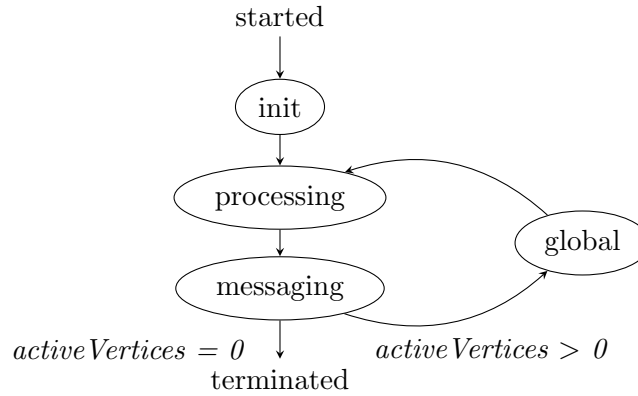


Figure 3.4: Global State Machine

be setting up initial variables and data structures in algorithm global memory. After *init* all the parallel *processing* phases are executed (distributed and parallel execution of c over all vertices) which are followed by an also parallel *messaging* phase. If at the end of the messaging phase the framework determines that there are no more active vertices the algorithm instance is switched to *terminated* state. Otherwise a *global* intermediate function (γ) is executed by the framework which allows mutation of the global memory and then the algorithm continues with another *processing* phase.

3.4.3 Vertex Local Computation

In the presented framework prospective users need to write graph algorithms from a vertex centric perspective. Processing is only possible in the local context of a vertex by creating an implementation of the *compute* function c and is always restricted to data that was recorded in a certain time window. The *compute* function is guaranteed to be called for every active vertex. Vertices are active as long as they have not voted to halt or if they have already voted to halt they become active again if they have received messages from $t - 1$. The *compute* function receives a list of all incoming vertex messages (x addressed to v) for the current vertex, a handle to memory to store vertex local information (called the *VertexContext*), a handle to memory to store algorithm global information (called the *SuperstepContext*), and a handle to the current vertex as parameters.

As already introduced before algorithms implemented in this framework can only access information from other vertices by sending and receiving messages. Vertices are able to send messages to any other vertex (including themselves) if the receiving vertices identifier is known, this means in the most cases messages are only sent along incoming and outgoing arcs to direct neighbours. In order to allow vertices to send and receive messages all computation is divided into steps which are also used for algorithm syn-

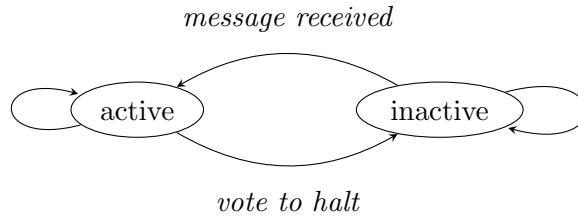


Figure 3.5: Vertex Local State Machine

chronization. The master node is managing the step counter and makes sure that any slot on any worker is executing step $t + 1$ only after step t has finished. If a vertex a sends messages to vertex b in step t the framework makes sure that the messages are routed in such a way that they are guaranteed to arrive at vertex b exactly in step $t + 1$. The user of the framework should get the impression that the algorithm for any particular step t is executed in parallel for all vertices (although this can only be true for graphs with a very small number of vertices).

3.4.4 Synchronisation and Halting

Each step also causes implicit synchronization in the execution of distributed algorithms. After a slot has executed one step of an algorithm on all vertices of its local partition, the slots make sure to complete routing of all messages to the corresponding receiving partitions. After that they notify the master node that they have completed execution. Dependent on the global execution state, the master schedules the next execution step, cancels job execution (if an error has occurred), pauses job execution (if cluster reorganization is necessary), or marks the job as finished when the algorithm has run to completion.

In order to determine when an algorithm run has come to completion the vote-to-halt mechanism already described is in place. Figure 3.5 shows the local state machine which is used for each vertex. By voting to halt a vertex is marked as inactive for the current algorithm run. The vertex can only be activated again if it receives a vertex message in the next execution step. In each execution step only active vertices are considered and after each execution step the slots report the number of their still active vertices back to the master node. It is then trivial to calculate the total number of active vertices for a certain algorithm run and the master halts algorithm execution when this number reaches 0 (all vertices voted for halt).

In figure 3.6 the timeline of an example algorithm run is displayed. It is clearly visible that after the master node has instructed all workers to initialize they start with local computation. Once a worker is done with its local computation it immediately continues

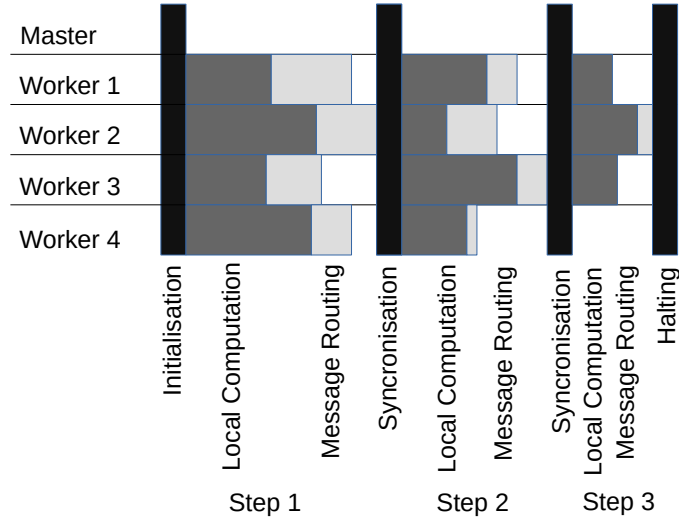


Figure 3.6: Schematic of an example algorithm run

message routing and reports completion of message routing back to the master. The master enforces a synchronisation barrier which means that all workers need to wait until all other workers have completed message routing for the respective step. If in any certain step (in figure 3.6 in step 3) local computation for all vertices has resulted in a vote for halt the master is halting algorithm execution.

3.4.5 Algorithm Initialisation and Global Computation

For storing global results and returning them back to the caller of a job the framework supports a volatile global context Γ . A local copy of the global context is available for any vertex and the same context is used in all processing steps t of a run. This global context is initialised once globally on the master node and then distributed to all worker nodes participating in the computation. To perform initialisation framework users are able to implement the κ (*init*) function of an algorithm which is being executed just after creation of the global context on the master.

During local execution of c for the vertices of a slot the global context Γ is not synchronised which means that the individual partitions are operating on local copies that, after a certain step t has finished, will differ from other copies on other partitions. Only after step t has finished executing the individual copies of Γ are sent back to the master node where they get merged together. Since merge conflicts will occur during this process, framework users have to specify merge strategies to be applied to certain fields of the Γ . Such strategies might be simple numerical functions such as *min* and *max*, but can also be more complex such as merge strategies that automatically merge sets

of data together. Moreover users are able to implement their very own merge strategies if the framework-provided ones are not sufficient.

After all workers have reported back from local computation and the data in Γ was successfully merged the *global* function γ is called on the master node. The function operates over Γ , the current execution step, and the number of active vertices as parameter. This allows framework users to reorganise data in the global context Γ which would otherwise be difficult to implement without a *global* function.

3.4.6 Multiple Phase Algorithms

As introduced earlier the framework presented in this paper also supports the execution of algorithms in multiple phases. Such that important graph algorithms that are traditionally built from several phases such as the maximum flow [100] and clustering algorithms [19] can be formalised as a single, contiguous algorithm rather than executing several loosely coupled methods. Aside the benefit of better logical structure in the algorithm, this opens up the possibility to pass intermediate and global results from one algorithm phase on to the next phase. In practice this means that intermediate results which at large-scale (i.e. vertex-centrality for every vertex) are difficult to gather in global storage can continue to live in intermediate memory directly on the compute node where this data is needed again.

The multiple phase algorithm mechanism is implemented in this framework by allowing for special algorithm packages that wrap up a list of child algorithms that are executed in order. Child algorithms can again be packaged algorithms which effectively allows framework users to submit complete hierarchies of algorithms in a batch. During the execution of the functions *compute* (c), *init* (κ) and *global* (γ) the respective vertex memory for each vertex and the global memory block Γ are re-used such that results from prior algorithm phases are available in consecutive ones.

This eliminates the need for framework users to take care of the execution phases by themselves and allows for simpler algorithm formalisation for algorithms like maximum flow which often consist of a forward tracing phase and a backward tracing phase. Further this allows to apply statistical functions to the results of algorithm runs, and simply to submit many algorithms in a batch that otherwise would have been submitted to the cluster consecutively.

3.5 Triggered Algorithm Execution

The aim of this framework is to be used in scenarios where ever changing networks are to be analysed. The focus on the temporal aspect obviously leads to the requirement of continuously analysing data. Examples for such scenarios would be network analysis which shall be performed on e-mail log files in near real-time. In this case it is desirable that the framework itself is aware of changes in data and automatically runs a pre-defined set of algorithm.

The framework allows this through a mechanism called triggered algorithm execution. The temporal data stored in the system is already allocated in time-slots according to a user-defined time resolution i.e. hours. When continuous data processing is required triggers for each n -th completion of a time-resolution window can be configured. Whenever a time-slot is elapsed the framework automatically checks whether or not new data was added to the last time-slot and if so executes a user-defined set of algorithms on this data.

3.6 Fault Tolerance and Reorganisation

As already discussed in section 3.2 partitioning functions p not only compute a primary partition number for any vertex added to the system but can also be used to compute secondary and backup partitions for each vertex. This secondary partition is used for recovery from node faults. In real implementations, depending on an applications requirements the number of copies to keep for each vertex can be configured. Independent of the number of backup copies configured the following regime applies.

For the mere data partitioning the backup partitions are used to host exact copies of vertices. In theory there are several different strategies that can be applied when creating the backup copies, mainly depending on which level of consistency between the master copy and the backup copies needs to be achieved.

- When full consistency between all copies is required the master node can wait for all worker nodes to acknowledge successful write operation on every vertex profile (the master and the backup copies). In this case all copies are the same at all times, however, write performance will be slow because positive feedback from all backup copies need to be awaited.
- In cases where more than one backup copy for every vertex exists a more relaxed regime with quorum can be used. In this case the master will instruct the primary

partition and a certain number of backup partitions to execute a given write operation. The master will consider the write successful as soon as the primary partition and more than a half of the backup partitions have confirmed the write. In case of node failure and reorganisation conflicts might be the case. For each vertex-conflict a quorum will need to decide which copy is correct.

- Finally, in scenarios where consistency is no primary goal all write operations can complete as soon as the primary partition confirms successful write with the risk that data is lost due to failed write operations on backup partitions or inconsistencies that cannot be resolved after primary partition failure.

The framework uses keep-alive messaging between the master node and the worker nodes to detect node failures. In case of a failure a reorganisation plan takes action. In a first phase all currently executing algorithms are paused for reorganisation. After that every slot on every worker node inspects their local partition. For each vertex in the partition the partitioning function p is used to compute the primary and all the backup partitions. If for any given vertex the computed partition identifiers are not compliant with the partition the vertex is currently placed then a migration process for this vertex is started. The migration process moves the vertex to its new target partitions and makes sure that old copies are deleted from the now wrong partitions.

Similar mechanisms can be applied for intermediate results stored on worker nodes. Intermediate results are either stored in vertex local memory or in global context Γ . On master node failure the global context is lost and all algorithm execution needs to start over. The intermediate vertex local results however can get copied to backup partitions after successful completion of each superstep. This allows to recover from worker node failure and to continue computation through resetting the current superstep. From the previous description it becomes clear that for each vertex a copy of the vertex local memory needs to be created at the beginning of each superstep this process is called checkpointing.

Since checkpointing and creating backups of vertex local memory to backup partitions (depending on the executed algorithm) may pose severe memory and runtime overhead these features are optional.

3.7 Selected Example Algorithms and their Processing

To illustrate how the framework can be used by prospective developers selected algorithms implemented in the framework are discussed in the following sections. A

reference implementation (as discussed in chapter 4) for this framework was created in Java. The algorithms described in this section were tested against this reference implementation and thus their listings are also in the Java programming language. However, it is important to point out that the framework does not strongly depend on Java specific features or the accompanying API such that other implementations of the framework can be created in other programming languages.

For the code listings given in the following sections usually only the method *execute* is given which is the function *c* which is to be executed for every vertex. Boilerplate code i.e. the class, field declarations, package imports, exception handling etc. are omitted for better readability.

The presented algorithms were later used to test-drive and evaluate the framework in applications of social network analysis and the analysis of web data.

3.7.1 Max- Min- Vertex Degree

In many applications the vertex degree also referred to as degree centrality (see section 1.3.2.1.1). Plays an important role, however absolute values for these are difficult to compare such that the absolute values are often normalized. For normalization aggregates of the vertex degree such as their minimum or maximum need to be computed.

Aggregates over vertex degree also have other interesting applications especially when computed over a temporal graph. For instance it is an indicator for how a social network evolves if the trends for maximum vertex degree or average vertex degree are observed.

Listing 3.3: MaxDegree

```

1  private void execute(List<VertexMessage> messages,
2      VertexContext vertexContext,
3      SuperStepContext superstepContext,
4      Timeframe timeframe, Vertex vertex) {
5      float degree = myDegree(vertex, timeframe);
6      float currentGlobalMaximum =
7          (Float) superstepContext.get(GLOBAL_MAXIMUM);
8      if(degree <= currentGlobalMaximum) {
9          voteToHalt(vertexContext);
10         return;
11     }else{
12         superstepContext.put(GLOBAL_MAXIMUM, degree);
13     }

```

```

14  }
15
16  public Map<String, Reducer> getReducers() {
17      HashMap<String, Reducer> reducers =
18          new HashMap<String, Reducer>();
19      reducers.put(GLOBAL_MAXIMUM, new MaxReducer<Float>());
20      return reducers;
21  }

```

The algorithm for the global maximum vertex degree implemented in Java can be found in listing 3.3 (minimum degree follows the same strategy). The listing covers two methods (1) *execute* which is the implementation of the vertex local function c and, (2) *getReducers* which defines the merging behavior of variables in global context Γ . On line 5 in the listing a helper function *myDegree* is used to determine the vertex degree of the current vertex. If this local vertex degree is below an already known global maximum then the vertex votes for halt (line 9) . Otherwise the vertex will assume that it itself is the vertex with global maximum and will write its own vertex degree in global context Γ .

Since the algorithm operates on a local copy of the global context Γ a merge strategy for the maximum vertex degree needs to be defined. This is done through implementing the function *getReducers*. In the listing on line 20 a max aggregation strategy (reducer) is defined to use the maximum value for *GLOBAL_MAXIMUM* during the merge process.

This algorithm has a rather atypical implementation since the global context is used for communication instead of using the messaging function m to send vertex-to-vertex messages. The cluster will show only short activity for all vertices in the system and will immediately halt.

3.7.2 Shortest Path

Another algorithm often used in applications of graph processing is the computation of the shortest path between any two vertices. The computation of all shortest paths in a graph for instance is the basis for calculating betweenness centrality and is a fundamental metric used in graph clustering.

In the following a distributed temporal implementation of a shortest path algorithm is discussed. This algorithm takes the vertex ids of the start vertex and end vertex as its

input and uses the Pregel-style processing framework to filter the shortest path from all possible paths. As clearly visible in the source listing in 3.4 there are two different branches a single vertex in a single iteration can take in this implementation.

Seeding (see lines 13 to 24) is executed on the first iteration in the context of the start vertex. This code branch creates a new *PathLength* object which is a simple Java object to hold the already visited path components and their overall length. This object is wrapped up in a message and sent to every neighbor of the current vertex through all out-going edges.

Intermediate Vertex: On all other iterations of the algorithm all other vertices in the graph will iterate over all received messages from other vertices. Once a vertex receives a message a first check will determine if the current vertex is the end-vertex (line 34) of the path specified in the received message. If this is not the case the current vertex must be an intermediate vertex. The vertex will check whether or not it is already in the given path. If it is already on the path a loop was detected (see line 42) and no further action is required. Otherwise the current vertex will add itself to the *Path* of the *PathLength* object and continue by propagating the modified *PathLength* object to its neighbour vertices along outgoing edges (lines 43 to 52).

End Vertex: If the current vertex is the end-vertex of the path then a check in global context will be made to figure out whether or not the current path is shorter than a already found shortest path, and the last global result will be overwritten by the new *PathLength* object.

Inactive: The last branch in the listing (line 57) is executed by all inactive vertices. Those that did not receive any messages and are neither the start nor end vertex of the path. In these cases the algorithm votes for halt such that once all messages in the cluster have been processed the algorithm runs to completion.

Listing 3.4: ShortestPath

```

1  public void execute(List<VertexMessage> messages ,
2      VertexContext vertexContext ,
3      SuperStepContext superstepContext ,
4      Timeframe timeframe, Vertex vertex) {
5      long startPathVertexId =
6          (Long) superstepContext.get(PATH_START);
7      long endPathVertexId =
8          (Long) superstepContext.get(PATH_END);
9      boolean seeded = false;
10     if("true".equals(vertexContext.get("Seeded"))) {

```

```

11     seeded = (boolean) vertexContext.get("Seeded");
12 }
13 if(!seeded && vertex.getId() == startPathVertexId) {
14     if(vertex.getOutEdgesReading() != null) {
15         for(Edge out:vertex.getWeightedOutEdgesReading(
16             "any", timeframe)) {
17             PathLength pl = new PathLength();
18             pl.start = startPathVertexId;
19             pl.end = endPathVertexId;
20             pl.visited = new Path();
21             pl.visited.add(pl.start);
22             sendMessage(out.getTarget(), pl);
23         }
24     }
25     vertexContext.put("Seeded", true);
26 }else{
27     if(messages.size() > 0) {
28         int shortestPath = Integer.MAX_VALUE;
29         if(superstepContext.get(SHORTEST_PATH) != null) {
30             shortestPath = (Integer) superstepContext.get(
31                 SHORTEST_PATH);
32         }
33         for(VertexMessage m:messages) {
34             PathLength pl = (PathLength) m.getBody();
35             if(pl.end == vertex.getId()) {
36                 if(pl.visited.length() < shortestPath) {
37                     superstepContext.put(SHORTEST_PATH,
38                         pl.visited.length());
39                     superstepContext.put(SHORTEST_PATH_DESC,
40                         pl.toString());
41                 }
42             }else{
43                 if(!pl.visited.contains(vertex.getId())) {
44                     if(pl.visited.length() < shortestPath) {
45                         pl.visited.add(vertex.getId());
46                         if(vertex.getWeightedOutEdgesReading("any",
47                             timeframe) != null) {
48                             for(Edge out:vertex.
49                                 getWeightedOutEdgesReading("any",
50                                     timeframe)) {
51                                 if(out.getTarget() != m.getSourceId())

```

```

48         sendMessage(out.getTarget(), pl);
49     }
50 }
51 }
52 }
53 }
54 }
55 }
56 }else{
57     voteToHalt(vertexContext);
58 }
59 }
60 }

```

This algorithm is a path following implementation that will show a behavior of having only one active vertex in the beginning that changes rapidly to a very active cluster once the initial seeding has resulted in a state where many copies of *PathLength* objects are in transit. The load will then go down as the algorithm converges towards finding a result.

The implementation given in 3.4 will not halt for queries where the start and end vertex are on different communities in disconnected graphs and moreover for queries where the given end-vertex is non-existent. For these cases a simple upper boundary on the path length can be used to make sure that also in these cases the algorithm runs to completion.

3.7.3 Page Rank

The next algorithm discussed in this section is *PageRank* [89] which was famously used (and with many modifications still is) at Google to drive the ranking algorithm for their web graph. A stripped down implementation of its compute function implemented in the presented framework is given in listing 3.5. As discussed in the original paper PageRank is an iterative process that assigns a rank to each vertex in a network. The algorithm initialises the rank for each vertex with a fixed constant, in our case 0.25. After initialisation each vertex broadcasts its own rank multiplied by a damping factor (in our case 0.85) and divided through the number of neighbours to its neighbors (see listing lines 9-19). For any other iteration t of the algorithm each vertex sums up the rank values received from its neighbors in iteration $t-1$ and sets the sum as its new rank and broadcasts this rank with exactly the same formulae as used during initialisation process (see listing lines 21-42). The algorithm votes to halt if a number of maximum

iterations was reached (listing lines 5-8) or does so implicitly if only little change to the values is detectable defined by a swing threshold.

After a certain amount of iterations the algorithm converges towards an optimal *PageRank* distribution which can be used to sort pages in search results. Pages or vertices which have a higher *PageRank* have more inbound links from other highly valued pages compared to other with only few inbound links from perhaps lower ranked pages. It is quite save to assume that those pages with high *PageRank* are more important.

Listing 3.5: PageRank

```

1 public void execute(List<VertexMsg> messages,
2   VertexContext vertexContext,
3   SuperStepContext superstepContext,
4   Timeframe timeframe, Vertex vertex) {
5   if(this.getStep() >= PageRank.MAX_ITER) {
6     voteToHalt(vertexContext);
7     return;
8   }
9   if(this.getStep() == 0L) {
10    setPageRank(vertexContext, PageRank.INITRANK);
11    Collection<Edge> outEdges =
12    vertex.getWeightedOutEdgesReading("any");
13    int numberOfOutEdges = outEdges.size();
14    float outRank = (pageRank * PageRank.DAMP) /
15      numberOfOutEdges;
16
17    for(Edge out : outEdges) {
18      sendMessage(out.getTarget(), outRank);
19    }
20  }else{
21    if(messages.size() > 0) {
22      float changedby = 0.0f;
23      float sumIncomming = 0.0f;
24      for(VertexMessage m : messages) {
25        Float incomming = (Float) m.getBody();
26        sumIncomming += incomming.floatValue();
27      }
28      changedby = setPageRank(vertexContext,
29        sumIncomming);
30      if(changedby >= PageRank.SWING_THRESHOLD) {
31        Collection<Edge> outEdges =
32          vertex.getWeightedOutEdgesReading("any");

```



```

32     float pageRank = getPageRank(vertexContext);
33     float numberOfOutEdges = outEdges.size();
34     float outRank = (pageRank * PageRank.DAMP) /
        numberOfOutEdges;
35     for(Edge out : outEdges) {
36         sendMessage(out.getTarget(), outRank);
37     }
38 }
39 }
40     voteToHalt(vertexContext);
41 }
42 }

```

Compared to the other algorithms discussed in this chapter the *PageRank* implementation is the perfect fit for the distributed computing model discussed here. Compared to the other algorithms *PageRank* have all vertices actively contributing to computation until algorithm halt.

3.7.4 Label Propagation Community Detection

The final algorithm discussed in this section is an implementation of a simple label propagation community detection algorithm originally developed by Raghavan et al. [94]. It is known to provide good quality communities in near linear time. Although the initial design of the algorithm is a synchronous process of neighboring vertices exchanging community labels, the very same algorithm can be implemented as an asynchronous process and thus fits the Pregel-style distributed computing paradigm very well.

Initially the algorithm assumes, that every vertex v lives in its own community. The community label assigned to each vertex is the vertex identifier. In the case of *DynamoGraph* this would be a long number (see figure 3.7a for the initialized state of a simple graph). In each consecutive iteration of the algorithm each vertex observes the community label of its neighbours and takes over the label that most of its neighbours currently hold. Variants of the algorithm also consider the edge weight. In the case that multiple community labels are candidates (i.e. all community labels occur exactly once) the smallest community label is chosen.

In figure 3.7b the graph is shown after the first iteration of label propagation. It is already visible that in the left community the label 0 was chosen often and in the right community the label 6. So it comes with no surprise that in the consecutive executions

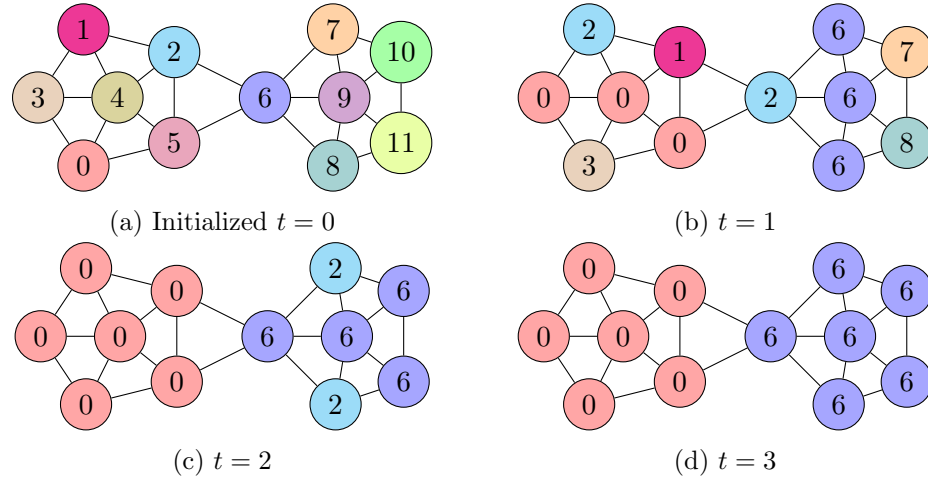


Figure 3.7: Four iterations of the label propagation executed over a simple graph with clear community structure

$t = 2$ and $t = 3$ (as shown in figures 3.7c and 3.7d) the labels 0 and 6 push through and two communities are formed.

In the given example the algorithm converges towards good communities in only three iterations. It is also clear that if the algorithm was to continue to execute further iterations, the community labels would remain unchanged. The communities found in this graph also refer to the communities a human observer would detect.

The presented algorithm has been shown to give good results on sparse graphs as found in popular application areas such as social network analysis and web-graph analysis.

Listing 3.6: Label Propagation Community Detection

```

1 public void execute(List<VertexMessage> messages,
   VertexContext vertexContext, SuperStepContext
   superstepContext, Timeframe timeframe, Node node) {
2   if(this.getStep() == 0L) {
3     this.setVertexLabel(vertexContext, node.getId());
4     propagateVertexLabel(node, vertexContext);
5   }else if(this.getStep() >= MAX_ITERATIONS){
6     node.getKeyValueWriting(timeframe.getStart()).put(
7       CLUSTER_ID_ATTRIBUTE, this.getVertexLabel(
8         vertexContext));
9     voteToHalt(vertexContext);
10  }else{
11    Map<Long,Integer> count = new HashMap<Long,Integer>();
12    for(VertexMessage m : messages) {
13      Long clusterId = (Long) m.getBody();

```

```

12         if(!count.containsKey(clusterId)) {
13             count.put(clusterId, 1);
14         }else{
15             Integer currentCount = count.get(clusterId);
16             currentCount ++;
17             count.put(clusterId, currentCount);
18         }
19     }
20     Long maxClusterId = null;
21     Integer maxClusterCount = 0;
22     for(Long clusterId : count.keySet()) {
23         if(maxClusterId == null ||
24             count.get(clusterId) > maxClusterCount) {
25             maxClusterId = clusterId;
26             maxClusterCount = count.get(clusterId);
27         }
28     }
29     if(maxClusterId != null) {
30         setVertexLabel(vertexContext, maxClusterId);
31     }
32     propagateVertexLabel(node, vertexContext);
33 }
34 }
35
36 private void propagateVertexLabel(
37     Node node, VertexContext ctx) {
38     Collection<Edge> outEdges = node.getOutEdgesReading();
39     Collection<Edge> inEdges = node.getInEdgesReading();
40     List<Long> candidates = new ArrayList<Long>();
41     if(outEdges != null) for(Edge e : outEdges) {
42         if(!candidates.contains(e.getTarget())) {
43             candidates.add(e.getTarget());
44         }
45     }
46     if(inEdges != null) for(Edge e : inEdges) {
47         if(!candidates.contains(e.getSource())) {
48             candidates.add(e.getSource());
49         }
50     }
51     for(Long l : candidates) {
52         sendMessage(l, this.getVertexLabel(ctx));
53     }

```

54 }

In listing 3.6 a naive implementation of the label propagation algorithm is presented. Each vertex in iteration $t = 0$ reads its own vertex id and stores it in vertex local volatile memory which is abstracted by the method call `setVertexLabel()`. This behavior relates to the initialisation procedure discussed before (see lines 2-3 in listing 3.6). Immediately after initialisation has completed the vertex sends its current community label to all of its neighbors for the first time. This is done in the subroutine `propagateVertexLabel()`.

The method `propagateVertexLabel()` (see lines 36-54 in listing 3.6) collects all edges (in and out) of the vertex and stores the vertex id of the neighbor connected through these edges in an array called `candidates`. The algorithm then sends the vertices' current community label to all these neighbour candidates through a message passing call (see line 52). Obviously depending on the exact use-case the label propagation can consider the edge-weights, ignore certain edges depending on arbitrary edge attributes, and moreover only work over out- or in-coming edges to follow the structure of a directed graph. Methods for these alternatives were created and tested but are not given in the written work for better readability.

In consecutive iterations of the algorithm community labels from neighbors are received (see lines 9-32) for processing. The algorithm counts the individual community labels received from neighbours using a `HashMap`. It then proceeds to find the label received most often which is the new community label for the vertex. After a new community label was determined again the method `propagateVertexLabel()` is used to distribute this label among the vertices' neighbours.

After a user defined number of iterations (see constant `MAX_ITERATIONS` and code from lines 5-7) the algorithm copies the community label from the volatile vertex local memory which is only available during algorithm execution to a vertex attribute and votes to halt algorithm execution.

3.8 Query Methods Implemented as Supersteps

A special form of algorithms to be run against the framework are queries. Queries can be implemented using the mechanisms described in this chapter, they are just special forms of distributed algorithms which can have a generic implementation since the features can be used in many different applications.

Queries are often used from client applications to interact with the distributed cluster. Assuming an application for social network analysis a common task for its users will be to visualise and interact with (parts) of the analysed graphs. Let us assume for a certain metric such as *PageRank* the highest ranked vertices and their neighbors are to be visualised. In this case the client application will run queries against the existing dataset to retrieve graph data from the backend system.

3.8.1 Single Vertex Retrieval

The simplest query one can think of is finding a certain vertex by its known vertex identifier. This can be easily implemented by writing a *compute* function c that finds the vertex with the known identifier in the local partition of the graph. The result of this query will be the full vertex profile of the found vertex.

In section 3.1 the use of temporal maps as the foundation layer for data storage in this framework was discussed. From this section it is also clear that a temporal graph is a map of temporal maps where the vertex identifiers can be used for vertex lookup. In fact a faster implementation to retrieving a single vertex from the system is embedded in the framework. Single vertex queries are handled directly by the framework through directly looking up the vertices in the distributed partitions. This results in better performance at the cost of not being able to integrate with a more generic implementation as discussed in the next section.

3.8.2 Queries for Vertex Attributes

A further query available in the framework is querying for any arbitrary attributes of vertices in a certain time-frame. The query is able to describe filtering against any attributes and hierarchies of attributes found in the temporal map. As shown later in reference implementation later in chapter 4 this functionality can be delegated to JSON query languages such as JSONPath ³.

Mainly the following filtering mechanisms are provided:

Comparison and Equality: For any arbitrary attributes equality can be checked i.e. the attribute *name* could be searched for a specific name string. For numeric values also the typical comparison operations (equals, greater, greater-equals, less, less-equal) can be used.

³JSONPath: <https://github.com/jayway/JsonPath>

In-Operator: Collections can be checked with an in operator for inclusion or exclusion of items with certain attributes.

For these queries naive implementations with a function c that gets executed over all vertex profiles can be found. The result of these queries will be a list of vertices to which the given attribute restrictions apply. Since in practice these result lists can grow very big in size (imagine a query without any restrictions returning the complete graph) limitations on the result size can be applied and are default.

3.8.3 Network Query Algorithms

Finally, also implemented through a generic version of the *compute* function c network query algorithms can be formalised. A network query algorithm is looking up a single vertex in the graph and returns it together with its neighbours. The query can retrieve neighbours up to a certain depth.

The query is implemented in two phases (1) the start vertex is found which then generates a depth message that gets sent to all of its neighbours (filtered by incoming and / or outgoing edges and their type), (2) every vertex that receives a depth messages adds itself to the result set reduces the depth counter by 1 and sends a new depth message to all its neighbours, this is continued until the depth counter reaches 0. All other vertices vote to halt.

3.9 Summary

In the last sections a data-model for distributed temporal graphs along with a Pregel-style processing mechanism was presented and discussed in detail. It was illustrated how the concept of temporal maps are useful in modelling a temporal graph and how this temporal graph can be partitioned over several computing instances. Different processing paradigms in parallel and distributed computing were briefly recapitulated and the case for compute aggregate broadcast models was made.

The presented processing model uses a vertex centric computing strategy where a central *compute* function c is called in the local context of every vertex in the graph. Since direct memory access is not possible in this paradigm vertex-to-vertex communication mechanisms based on message passing are used for synchronisation and data exchange. The presented processing model runs in iterations until processing at all vertices has voted to halt. These iterations cause implicit synchronisation barriers.

Finally some typical graph algorithms and their possible implementation on top of the presented framework were discussed. A concrete reference implementation of the framework itself, several graph algorithms, client applications and evaluation are to be found in the following chapters.

Chapter 4

Reference Implementation of the Distributed Processing Framework

In order to show the technical feasibility of the aforementioned approach, and to test different algorithm execution strategies, partitioning schemes and cluster configurations a prototypical implementation was created. The prototype provides a real world implementation of the distributed temporal graph processing framework presented in chapter 3. To make the prototype usable for testing, evaluation, and real world applications eventually also other components had to be added to the system. Such that the prototype also provides a full stack of temporal graph storage components, and user interface geared towards interactive social network analysis. The chapter at hand will describe all layers of the software stack in greater detail and will highlight how they contribute to the overall goal of large scale distributed temporal graph computing.

The remainder of this chapter is structured as follows. Section 4.1 describes additional requirements for a real world implementation of the framework. Section 4.2 gives a brief overview of the general architecture and its components. In section 4.3 mechanisms for distributed configuration and coordination and their implementation are discussed. The section 4.4 covers how worker and master role nodes use intra-cluster communication to route and process data and command packets and section 4.5 describes how this mechanism is used for distributed algorithm execution. Section 4.6 discusses basic functionality of the Client API and lists the most important API calls. One of these API calls provides dynamic Java code loading which is explained in greater detail in section 4.7. Section 4.8 discusses the implementation of monitoring sensors. Section 4.9 provides information on the implemented service for data persistence. Finally 4.10 sheds light on some possible improvements and section 4.11 concludes this chapter. This chapter is accompanied by appendix A which contains in depth details on the tooling used during implementation, exact version numbers of all tools, where and how to obtain the source code of this prototype implementation including an online reference to up to date implementation details.

4.1 Extended Requirements

Besides the functional requirements discussed in section 1.5 and the ideas of an ideal temporal graph processing system presented in section 1.6 several other requirements need to be met once a prototype implementation is to be created. The following sections will describe these requirements of a real world implementation and provides pointers to their realisation within the reference implementation.

4.1.1 Configurability

A first very important requirement is that the system is configurable. This requirement is reflected mainly in two dimensions. First of all the system needs to be able to detect changes to its topology and to automatically reconfigure if such changes (i.e. compute nodes are added or removed from the system) occur. This is strongly related to the requirements of scalability and fault-tolerance (which is discussed in detail in section 4.3.3).

Second, system administrators need to be able to set basic system configuration parameters. Such parameters include the number of parallel threads to be executed on a single compute node, the location of log files and settings for debugging algorithms. Some of these settings (i.e. log and debug settings) should even be modifiable during runtime. This requirement is tackled by a Java property file. This file is accompanied by a caching in-memory implementation that reads this file on-demand and caches the values in the file for a configurable amount of time in memory. Some configuration parameters are marked as "fixed" in the source code and are only read once during the first use of the parameter.

4.1.2 Scalability

As data sets to be processed in this systems tend to grow very large in size and moreover will keep growing over time, this processing system needs to be able to accommodate to this and also scale. The point was made in previous section 3.3 that vertical scaling (i.e. adding more processors and memory to a single computer) has its limitations. On the one hand one can hit boundaries set by technical restrictions, and on the other hand vertically scaled machines as those used for high performance computing tasks are custom built and have a highly leveraged price point. The framework discussed in this thesis allows to distribute data over many different machines which is called horizontal

scaling. If storage, and processing capacity is running out new computers (compute nodes) can be added to the system to accommodate to these new requirements.

Adding (and also removing) nodes from the system will require dynamic reconfiguration of the system. Control instances need to be made aware of the new compute nodes to being able to facilitate them and depending on the graph partitioning scheme currently in action the data needs to be rebalanced. Details about the registration and de-registration of worker nodes can be found in section 4.3.2.

4.1.3 Fault Tolerance

Strongly linked with the scalability is the requirement for fault tolerance. With adding number of nodes to a computer the probability that any of these nodes fails rises. Thus the system needs to be able to react to node failure.

This requirement has two perspectives, first of all the data stored on node will become unavailable to the system. This is why the framework's partition manager supports a primary and a backup partition where a copy of all graph vertices is stored. On the other hand in an active distributed computing process also computation will break and needs to be rescheduled if a compute node fails. In section 4.3.3 the actual implementation of node fail-over detection and recovery are discussed.

4.1.4 Multi-Tenancy

As this system can be used to analyse graph data on a distributed compute system setup is complex and needs a little bit of planning ahead. This means that users of this framework will not be willing to roll out an individual installation for each data set to be analysed in the framework. It is rather the case that a single installation of the framework is used to host several datasets possibly from different tenants accessing the same potentially cloud based system. This leads to the requirement that multiple tenants can host their data sets on a single installation of this system and share the distributed computing capacities for their different analysis jobs.

This requirement is mainly addressed by the framework implementation through the use of what is called a *namespace*. Namespaces can be compared to tablespaces in traditional relational database systems however in this case they are used to distinguish between individual graphs. A namespace is a name unique to a single installation that names a graph dataset. All graph related tasks (adding/modifying vertices, executing

algorithms, etc.) are always executed in the context of such a namespace such that many datasets can co-exist on a single installation of the framework.

In commercial cloud scenarios multi-tenancy also means that a system needs to have sophisticated and fine-grained access regulations that allow to describe which users are allowed to access which parts of the system. Moreover in multi-tenant systems isolation between the individual tenants is usually highly preferred. Since the prototypical nature of the reference implementation described here is mainly used in scientific on-premise scenarios access regulations and isolation were not implemented.

4.1.5 Monitoring

When algorithms are executed in distributed environments debugging becomes inherently hard. Traditional mechanisms of step-debuggers no longer hold since it is not feasible to step-debug on many distributed (potentially hundreds) compute nodes in parallel. Thus ever more important is it to being able to monitor a system during varying load scenarios. System monitoring allows framework users to keep track of certain basic system parameters such as memory usage, CPU load, disk usage. Further the core of this system, the execution of massively parallel algorithms need to be closely monitored. For each run the start, and end time are to be recorded and all also the timings of all intermediate steps in-between. Further, details about the algorithm such as the number of vertices active per step must be recorded.

In the reference implementation monitoring data is collected by background threads running on all nodes in the cluster and sent in configurable time intervals to the master node which keeps short back-logs of this data in memory. Monitoring data can be picked up through API calls and thus be visualised and analysed by client applications. Monitoring was not a key aspect when creating this framework. The implementation of a web based monitoring framework for DynamoGraph thus was outsourced to a supervised masters thesis titled *Monitoring and Benchmarking Toolchain for the DynamoGraph Framework* [113]. In section 4.8 the extensions of the framework for monitoring and the inner workings of the sensors are discussed.

4.1.6 Modularity

Since this framework is developed as part of research agitations many different aspects of the framework are to be analysed. This means re-configuration, experimentation with new code and comparing different implementations are the norm. This leads to the requirement that everywhere possible the contracts between system components

are to be designed as Java service interfaces and are implemented through configurable implementations. Examples for components for which this is true are graph partitioning algorithms, persistence implementations, inter-process communication, and object serialisation to name only a few.

4.1.7 Persistence

While not necessary for network analysis per-se, during use of this framework in real world applications the requirement for persisting data within the system becomes evident. Users in real world scenarios are not going to be willing to upload datasets over and over whenever a cluster restart occurred. This means that the system needs to be able to run in a mode where the graph data is not only available as an in-memory model but is stored on persistent storage and thus survives cluster restarts. In section 4.9 the different storage models implemented in the prototype are discussed in detail.

4.1.8 Dynamic Code Loading

In a real world distributed computing framework developers as the users of such a framework need a way of uploading their own user code to the system. The framework needs to be able to dynamically load user provided code and execute it within the context of the framework. Static code loading (and linking) is an impractical scenario since this would require cluster restarts on every user's change. In a distributed computing environment cluster restarts can be a very time consuming process which is to be avoided at all cost during day-to-day operation. In section 4.7 the implementation of a dynamic code loader for the JVM is briefly discussed. This loader is used by the framework to load, manage, and unload user code.

4.2 Architectural Overview

The aforementioned additional requirements lead to the implementation and selection of components that complement the basic processing framework. This leads to an updated architecture which is displayed in figure 4.1. The diagram shows a high level view of a DynamoGraph deployment. Depending on the use-case alternative deployments are possible i.e. in development scenarios all processes illustrated in the diagram could run on the same machine, on the contrary in very large scale deployments one could decide to run dedicated nodes for the supporting services (ZooKeeper and Cassandra).

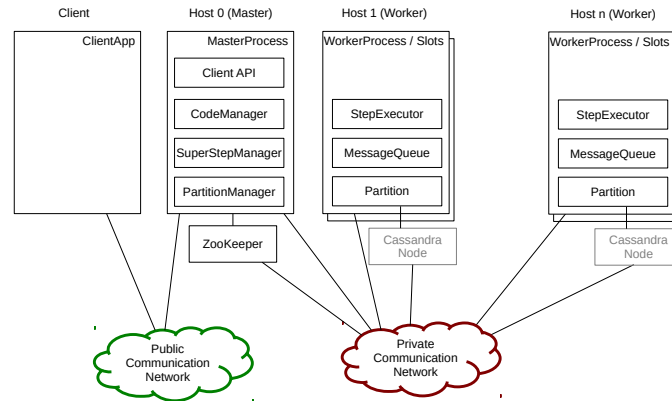


Figure 4.1: Architectural Overview of the Reference Implementation

In 4.1 a client application process is connecting through a public communication network to the master process. The client application is directly talking to the client API interface which is discussed in more detail in 4.6 for querying and manipulating data stored in the system, uploading and running graph algorithms, and retrieving monitoring data.

The *MasterProcess* is the component which runs on a dedicated master node. In practice the master node will often also act as a worker node since workload for this process is relatively low. The process uses several components worth mentioning:

CodeManager: This component is a service which allows framework users to upload code as Java JAR files. The service holds a registry of uploaded algorithms and is capable of registering code with worker nodes.

SuperStepManager: The SuperStepManager is the component responsible for code execution. Using the services provided by the CodeManager the SuperStepManager is capable of starting, pausing, and stopping distributed execution.

PartitionManager: Here the implementations of partitioning functions are living. For each namespace registered in the system a partitioning function is registered. The PartitionManager is able to compute the partition number for any given vertex and is able to distribute the partitioning function and partitioning table which is a mapping of partitions to worker nodes to the worker nodes.

In addition a *MonitoringService* (not listed in the figure) receives monitoring data from the worker nodes and keeps it in memory to be retrieved by client applications.

Alongside with the *MasterProcess* the ZooKeeper service responsible for distributed coordination is running on the same host as the master node. Technically communi-

cation with ZooKeeper is done through TCP/IP such that the service can also run on a dedicated host in larger setups. ZooKeeper provides a directory structure for configuration parameters. Operations on top of ZooKeeper provide synchronisation and locking-mechanisms that can be used to implement patterns often used in distributed applications.

Through a private communication network nodes running the *WorkerProcess* connect to the master. Starting up the *WorkerProcess* will in fact start a configurable number of parallel slots on these nodes. Each slot is an independent copy of all necessary worker services that only share the layer for network and database access. The most important services running for each slot are:

StepExecutor: This service takes instructions from a masters *SuperStepManager* to start, pause, and stop processing of supersteps. It contains a list of all executing algorithms alongside with global status. Moreover vertex local volatile memory is stored in this component.

MessageQueue: In the Pregel paradigm vertex-to-vertex communication is the main means of inter-process communication. The *MessageQueue* is responsible for keeping a buffer of outgoing and in-coming messages for each vertex. The component uses the partition table and partitioning function to route messages to the correct worker processes. The buffer provides better efficiency since multiple messages that need to get routed to a different worker can be sent in larger blocks.

Partition: The *Partition* service is the data storage and access layer of each worker node. It provides the functionality and in-memory data model for graph partitions stored on each individual worker. Optional storage backends can be used to make data stored in a partition persistent.

Worker nodes can optionally run storage backends. In figure 4.1 a Cassandra storage backend was added to the system. Cassandra database nodes can run alongside the *DynamoGraph* service on the worker nodes.

4.2.1 Big Data and Cloud-based Computation

Computing based on scalable cloud technology is today often used to compute problems that fall into the category of Big Data. The most popular definition of Big Data currently is given by NIST [76]. One faces a Big Data problem when one of the 3-Vs: Volume, Variety, and Velocity grow so big in size that the data becomes awkward to handle on a regular computer. This means data grows so big in size that it does

not fit on a regular single computer system for storage and processing, that it comes in such broad variety that regular analysis methodologies from business intelligence no longer hold, or that the data arrives at computing systems in a velocity that makes it impractical to directly compute on it.

Most solutions addressing Big Data today (aside the traditional high-performance computing arena) run on commodity hardware and use cloud based computing frameworks as an abstraction layer for computing infrastructure. This allows the data to be analysed to live in cloud-based scalable distributed data-storage and the computation be performed in cloud-based scalable computing resources. Without discussing concrete systems now (for which the interested reader is referred to chapter 2 related work) the idea inherent in most Big Data processing systems is, that instead of loading data from a storage system to a computing component, data is stored in a distributed fashion on a compute cluster. Algorithms are "pushed" down to the servers actually storing the data for computation on local partitions of the data.

From the descriptions given in the last sections it is clear that also the framework discussed in this thesis can be used to address Big Data problems. Users of the framework can store temporal graph data in the cloud, then formalise algorithms and send those algorithms to the compute cluster for distributed execution.

In cloud computing three basic service models were defined. These models are Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). IaaS provides virtual computing components through an service API, such that the tasks of provisioning computing resources such as storage, virtual machines, and network can be done in software. PaaS provides complete compute platforms to its users such that tasks such as installing frameworks and libraries on top of concrete (virtual) machines are outsourced to a service provider. And finally SaaS is a service model where users are consuming a software system (often through the web) which is completely maintained by the service provider.

The framework presented in this thesis is a heavy user of the IaaS paradigm. Numerous automatic provisioning scripts were created throughout developing the reference implementation. Complete cluster installations can easily be rolled on popular private cloud stacks such as OpenNebula and OpenStack, and on public cloud stacks such as Amazon AWS ¹. The automatic provisioning of the framework would make the leap of providing *DynamoGraph* as a PaaS system small. Details on the automatic provisioning are described in the evaluation chapter 6 (see section 6.3.3) where the provisioning is also used to run automated tests on an infrastructure cloud stack.

¹<http://aws.amazon.com>

4.2.2 Technological Decisions

As of writing this for data analysis, graph processing, and distributed systems in general a plethora of different base technologies are available. Programming languages, graph frameworks, databases, caching frameworks, and cloud solutions exist plentiful which makes technology decisions in this area very difficult. One may argue that the reference implementation could also have been implemented with a different programming language or on top of a different platform. However, this reference implementation was built on top of the Java VM and mainly in the Java programming language.

The main reason for this is that most popular Big Data processing frameworks available now are mainly written in Java. Some of these popular platforms are Hadoop², Storm³, Spark⁴, Giraph⁵, and GPS [96]. By using Java also for the implementation of DynamoGraph it was hoped to be able to reuse components from other popular processing stacks, to appeal to other Big Data developers since there is less overhead in learning a completely new programming environment, and finally to provide some basic compatibility with other systems as those named before.

Further, Java seems to provide a good trade-off between a compiled language and an interpreted language. Since Java is compiled to byte code which is then interpreted in the Java VM one gains the performance benefits provided by a compiled language and still is able to use features of interpreted languages such as dynamic loading and unloading of code as explained in section 4.1.8. Other related tools used during the development of the reference implementation, their exact versions, the Maven / Java projects created, and further implementation details are summarised in appendix A.

4.3 Distributed Configuration with ZooKeeper

Since the prototypical implementation is designed to run on top of cloud based environments also cluster configuration needs to use a cloud based configuration framework. Distributed cluster configuration poses special requirements since it is desirable that all nodes in the cluster see the same configuration information at any point in time. If this requirement can be met also distributed synchronisation mechanisms can be implemented on top of the configuration framework.

²Apache Hadoop: <http://hadoop.apache.org>

³Apache Storm: <http://storm.apache.org/>

⁴Apache Spark: <http://spark.apache.org>

⁵Apache Giraph: <http://giraph.apache.org>

In the DynamoGraph project distributed configuration is used as a naming directory for compute nodes participating in the compute cluster. The naming directory holds a list of all nodes part of the system and the TCP/IP ports used as their service ports. Further the naming directory holds information about the role of each node.

As a distributed configuration framework the Apache ZooKeeper services⁶ was chosen since it has been proven to be stable in several open source distributed computing projects such as Apache Hadoop and Apache Storm. ZooKeeper is implemented as a server application that provides a virtual directory structure to connecting clients. For each mutation in the directory structure and also the content of directories ZooKeeper guaranties that all clients see the very same information at any point in time. Further ZooKeeper provides simple mechanisms for conflict solving such as automatically enumerating directories that were created by different clients under the same name (called *sequences* in ZooKeeper), and asynchronous creation of directories such that success or failure is reported through a callback handler (called a *monitor* in ZooKeeper). Finally, ZooKeeper supports ephemeral directory creation modes which ensures that all data created by a certain client gets removed from the directory once the client disconnects (more precisely fails to continuously send keep-alive messages).

In DynamoGraph ZooKeeper is used to serve three purposes (1) it is the service directory for the compute cluster where general configuration information, and the IP addresses of the master and worker can be looked up, (2) the cluster system uses it for election of a master node (see section 4.3.1), (3) clients use it as entry point to the system to query the IP address and port of the DynamoGraph service API.

The service directory is implemented using two directories. In `/Master` the service endpoint for the current master node is stored, and the path `/Workers` contains a subdirectory for each worker node registered in the system. The worker node subdirectories contain pointers to the service endpoints for each worker. ZooKeeper supports several directory creation modes that affect how ZooKeeper reports changes to other clients connected to the service and querying information from a directory (in ZooKeeper this is called monitoring). For the service directories (master and workers) the `EPHEMERAL` creation mode is used. In this mode newly created directories are evicted from ZooKeeper once the client creating it gracefully quits or fails to send keep-alive packets. Other ZooKeeper clients monitoring a directory get notified about directories that get deleted such that nodes that get added and removed from the system can be handled properly.

New worker nodes added to the system are detected by the master node and lead to pausing current algorithm execution and asking the partition manager on the master

⁶Apache ZooKeeper: <http://zookeeper.apache.org>

node to rebalance partitions. Worker nodes vanishing from the system show similar behavior but will also re-schedule algorithm execution for algorithms that failed due to results missing from a failed worker. If the `/Master` directory is removed from ZooKeeper the remaining compute nodes try to recover and start a master election process (see section 4.3.1).

Finally ZooKeeper is also used for coordinated operations i.e. a command line client application for cluster management is able to store instructions in ZooKeeper that are to be executed by all workers. Through this mechanisms administrators are able to manually start an election process (through deletion of the `/Master` directory), gracefully remove a node from the cluster (waiting for all algorithms to run completion before removing a worker), and finally to shutdown the complete cluster through creating the `/Kill` directory in ZooKeeper.

4.3.1 Master Role Compute Nodes and their Election

In every DynamoGraph cluster there is a single compute node that acts as the systems master node. From a clients perspective the master is a single point of contact to the cluster system. It manages the currently stored graph, their partitioning over worker nodes, and the execution of temporal graph algorithms.

In general every node in a DynamoGraph cluster can hold the master role in the system. By default the master node runs both, the master and the worker-process. However, this behavior can be disabled through a configuration setting (`DESIGNATED_MASTER`). The master node gets elected through a protocol implemented on top of ZooKeeper. Here the `SEQUENCE` and `EPHEMERAL` directory creation modes of ZooKeeper are facilitated. The `SEQUENCE` mode acts in a way of automatically enumerating directories created with the same name. In the case of DynamoGraph the directory `/Election/sequenceNumber` is created by each client joining the cluster. ZooKeeper automatically enumerates this and will for example create `/Election/sequenceNumber.0006` for a particular node. The compute node that has the smallest sequence number will be elected master node.

Once a compute node was elected master it fires up all necessary master services and the service API to be used by clients and creates all necessary internal data structures to manage the cluster (i.e. a partition manager, a storage backend, a directory of active algorithms). It will then notify all other compute nodes in the system through creating the `/Master` directory which contains the internal service endpoint which worker nodes can use to contact the master.

After the master processes are running the master is in **Disabled** mode. Once the `/Master` directory shows up in ZooKeeper all worker nodes will start to initialize. They reconfigure their message queues and partition manager. After completion they start sending keep-alive packets to the master. The master switches to **Enabled**. In **Disabled** mode all algorithm execution and certain API calls (graph manipulation) are completely disabled. If the master process fires an exception an exception handler on the master node decides whether to completely kill the master process which will lead to starting the election process. If a new election process does not make sense (database connection failed, a configurable number of worker nodes failed at the same time, etc.) then the master switches to **Failed** which shows the same behavior as **Disabled**, currently the master cannot recover from **Failed**.

As explained earlier the directory for the sequence number and also the directory containing the masters service endpoint information are created in **EPHEMERAL** mode. Which means that the directory is deleted from ZooKeeper immediately if a client disconnects. This mechanism is leveraged to implicitly monitor the system for master node failure. All nodes in the system monitor the `/Master` directory, if it vanishes a new election process is started.

In real world tests it turned out that the workload imposed by the master node is in a range where it is sufficient to also run a worker node role on the same compute node with the master node.

4.3.2 Worker Role Compute Nodes

By default every node started up in a DynamoGraph cluster is started as pure worker node. During startup worker nodes query the ZooKeeper `/Master` directory to retrieve the internal cluster service endpoint. If no data could be retrieved the election process as explained earlier is started. If a master node was found and reachable (by sending a keep-alive packet) through the denoted service endpoint the newly started worker nodes proceeds with internal setup.

Since in modern computing hardware multi-core CPUs are the norm each worker node launches several processing threads to facilitate optimal resource usage. These threads are called *slots* in the system. Each is responsible for managing a partition of the graph and running algorithms on their local partition. First configuration files are read to determine how many parallel processes and thus partitions should be hosted on the worker node. All slots share some common infrastructure but aside from that they manage their local partition independently and execute code in their own thread. For administrators and developers to determine a good number of slots per worker node

the number of CPUs installed on the host machines is a good guess. This way each slot can run on a reserved CPU; first tests with overbooking CPUs by 150% also gave good results. The actual number of slots will need to be determined on a use-case basis. The current implementation of the prototype does not allow for slots to be changed dynamically. Changes of slots require coordinated restart of the complete cluster.

Slots share a common *PartitionTable* which provides the implementation of the partitioning function to determine on which worker and slot a certain partition is living. Further a component called **PacketRouter** is created during worker node start-up which is capable of routing messages locally or remote through the communication network. The component *StepLoader* provides code loading capabilities to all slots and is used by the *StepExecutor* to load user-provided Java code. Finally, a component called **NodeMonitor** collects monitoring data from the worker and all slots and submits it in batches and in a regular interval to the master node.

Slots themselves initialise two things (1) they create a **ModelNameSpace** which is the component that manages the local graph partitions of multiple namespaces assigned to the slot, (2) the slot starts a processing thread that waits for incoming packets and reacts on the encoded commands. After the worker process itself and all slots have successfully initialised the worker node creates a sequence directory on ZooKeeper named `/Workers/worker.XXXX` which contains the IP address and port numbers of the worker node. As the master node is monitoring `/Workers` for changes it gets notified about the added worker node and finalise worker node registration.

Finalisation, depending on the partitioning algorithm in use, can lead to pausing algorithm execution and reorganisation of the data. Assuming that the **PartitionManager** on the master node concludes that redistribution of vertices is necessary it computes a new **PartitionTable** and broadcasts the new partition table to all slots. The slots use the partition table to iterate over their local vertices and move all vertices that need to be migrated to a different partition to the new slot. This behavior can be omitted by using a Key-Value store as a persistence layer (see section 4.9.2) which is preferable in a real world scenario, in worst-case scenarios large volumes of data need to be transmitted through network links in order to apply a new partition schema.

After the cluster returns to a re-partitioned state the master node continues to accept new commands from client applications and can instruct worker nodes to execute data manipulation commands or to run superstep algorithms through the **SuperStepManager**.

4.3.3 Node Failure and Recovery

The last section discussed adding nodes to the system which is a normal operation during cluster startup and when users are up-scaling a compute cluster. On the other hand nodes can also be shut-down either during a down-scaling process, or because a software or hardware failure. Depending on the node different recovery scenarios can be run.

Currently three different methods are employed for detecting whether any node in the cluster failed (1) the logical communication links between the individual processes are implemented as TCP/IP socket connections, if they break node-failure is assumed, (2) a watch dog running on all nodes sends keep-alive packets at a configurable interval to detect broken network connections, and (3) the ZooKeeper directories `/Master` and `/Workers/worker.XXXX` are created using `EPHEMERAL` mode such that they get removed if the creating process died. All three methods are implemented in a single watchdog class called `SocketWatchDog` which notifies workers and master about node failures.

In general there are two situations that need to be handled (1) a worker node failed, (2) the master node failed. The latter in certain setups will mean that also the worker node running on the same computer failed.

In case a worker node failed the master node switches into `Disabled` mode and discontinues to take instructions from client applications. The master then sends a keep-alive packet to all worker nodes in order to determine which worker nodes are still in a healthy state. Depending on the replies it receives from the workers the partition table is updated (unless too many workers failed in that case it switches to `Failed` state). The new partition table is then broadcast to all the workers which as already discussed in the last section will lead to redistribution of vertices in the cluster. Now with the only difference that also the vertices currently in backup-partitions are taken into account such that data which was residing on the failed worker node is recovered from the backup partition.

After reorganisation has completed the master node switches back to `Enabled` state and thus continues to accept commands from client applications. The switch operation also notifies the `SuperStepManager` which in turn will restart all supersteps that did not run to completion.

4.4 Intra-Cluster Communication

Apart from basic configuration and coordination functionality ZooKeeper turned out to provide too high latency to be used as a mechanisms for node to node communication. Thus alternative forms needed to be found. It was already revealed in earlier sections that the current intra-cluster communication layer is based on TCP/IP socket connections with output buffers. TCP connections were chosen over UDP packets because TCP already provides state-full connections and error correction over the communication network. UDP packets would have had the disadvantage that almost all communication sent from one node to another would have needed some sort of acknowledgment mechanism to avoid errors originating from lost UDP packets.

Nevertheless the intra-cluster communication services were implemented as abstract classes such that the lower-level access to communication links can be implemented on top of a variety of different systems. The contract given by the service interface requires that two Java interfaces are implemented. The first interface is called **PacketHandler**, this interface is responsible for receiving incoming communication data which are encoded as **Packet** objects. All commands, and data transmitted in the cluster are wrapped up in **Packet** object or child objects of **Packet**. The **PacketHandler** also needs to implement a method called **getServiceEndpoint** which generates a network endpoint description for the local communication service. In the current implementation which is based on TCP/IP sockets this service endpoint description consists of the IP address of the host and the port that is used to receive incoming data. This information is used directly to feed the service directory in ZooKeeper. Other implementations of a network communication layer thus can encode their connection information in a compatible way (i.e. the URL of a message feed on a message queue system).

For the outgoing direction the abstract class **PacketSender** needs to be inherited. The method **send** needs to be implemented on top of the chosen communication layer. The **send** method receives the **Packet** and connection information of the target partition as parameter. In the TCP/IP implementation provided in the reference implementation connection pools are used to re-use the TCP/IP connections for optimal performance.

By default the framework assumes that a state-full communication network is used which is not losing any data and provides proper exceptions if network links break. However, first tests with state-less communication based on UDP were made. Thus the framework supports a mode where every sent packet is acknowledged by the receiver with an confirmation packet. This behavior can be enabled through a system setting called **CONFIRM_PACKETS**.

On top of the generic communication layer which is capable of sending objects of type `Packet` different types of payload can be implemented. These payload can be node to node commands (i.e. the master node instructing a certain partition to store a vertex), and vertex to vertex commands (i.e. during a superstep execution a vertex sends a message to another vertex). Technically speaking the `Packet` objects are Java objects and are serialized into byte data prior sending them through a communication link.

In the reference implementation also the serialisation process is implemented as a generic interface such that different forms of data serialisation can be tested. As the current default implementation the Java serialisation mechanism (in the reference implementation named JDK-serialisation) is used. However, other serialisations libraries such as the fast java serialisation drop-in replacement (FST)⁷ which claims to provide better performance compared to JDK serialisation, and Kryo⁸ another fast serialisation library famously used in many Big Data projects (Apache Storm, Apache S4) were added to the system for experimentation.

Further, a JSON serialisation mechanism based on the Gson library⁹ was added to the reference implementation. This implementation provides the advantage that during debugging the network traffic remains human-readable.

Framework users are able to select any of the predefined serialisation mechanisms by setting the configuration value `SERIALIZATION`. New mechanisms can easily be integrated by extending the class `CommandClient`.

4.5 Algorithm Execution

The main goal of the reference implementation was to cover the distributed temporal graph computation model which was explained earlier in section 3.4. As a short recap the framework is modeled after Pregel which is massively parallel processing strategy for graphs. Pregel was extended for some further functions as already discussed in section 3.4.1. At the core of Pregel is a *compute* function $c(v, \Gamma)$ which is executed for each vertex in the graph. The function signature receives the vertex v and a copy of the global memory Γ as parameters.

In the reference implementation the *compute* function c is reflected by a method in the abstract class `SuperStep`. This class provides an abstract method called `execute` which is to be implemented by framework users. Further the class provides two empty

⁷<http://ruedigermoeller.github.io/fast-serialization/>

⁸<https://github.com/EsotericSoftware/kryo>

⁹<https://github.com/google/gson>

methods `init`, and `global` which refer to the Pregel extensions for initialising global memory and performing global actions between each step.

The class further provides helper methods used during algorithm execution. Most importantly functions to start execution of the step on a partition in a particular namespace, functions that allow for vertex-to-vertex messages being sent and received in the `compute` function, and the `voteToHalt` method that sets the vote-to-halt flag for a particular vertex.

In the class `SuperStep` also an empty method `getReducers` is found. This method can be overridden by algorithm implementations in order to specify how certain attributes of the distributed copies of global memory need to be combined. This aspect of supersteps is discussed in greater detail later in this section.

For framework users to run an algorithm on top of the framework they need to provide a non-abstract implementation of a superstep through extending the `SuperStep` class. An example of a minimal superstep is given in listing 4.1. The superstep does not perform any operation in the `init`, and `global` method. In the `execute` phase the step immediately votes to halt which results in a superstep not performing any useful operation.

Listing 4.1: Minimal implementation of SuperStep

```
1 public class NoOperationStep extends Superstep {
2
3     @Override
4     public void global(SuperStepContext c, long stepT) {
5     }
6
7     @Override
8     public void init(SuperStepContext c) {
9     }
10
11    @Override
12    public void execute(List<VertexMessage> messages,
13                        VertexContext vertexContext, SuperStepContext
14                        superstepContext, Timeframe timeframe, Vertex vertex)
15                        throws SuperStepException {
16        voteToHalt(vertexContext);
17    }
18 }
```


For the framework being able to execute a superstep the step implementation needs to be loaded in the JVM executing the master and all worker nodes. Framework users can load custom load their supersteps into a running cluster as explained later in section 4.7.

In actually executing an algorithm two important components are involved. On the master node the class **SuperStepManager** manages the execution of all supersteps in the cluster, keeps track of their status and their global memory. On the worker nodes the class **Slot** is responsible for executing code in the context of a namespace and a particular partition.

When a framework user starts a superstep through a client API call (see section 4.6) a **SuperStepExecutionProfile** is created on the master node. This profile holds all important information for a superstep such as the fully-qualified name of the class implementing the superstep, the namespace the step is executed on and the time-frame that needs to be considered by the step. During creating of the **SuperStepExecutionProfile** also an instance of **SuperStepContext** is created which is the reflection of the algorithm global memory Γ and is an extension over a Java **HashMap**. The execution profile together with global memory is then registered at the **SuperStepManager**.

The **SuperStepManager** then uses a Java class-loader to load the superstep implementation class denoted in the **SuperStepExecutionProfile**. Once this load was successful the manager calls the **global** method on this class in order to allow the superstep to properly initialize global memory.

After initialisation has completed the **SuperStepExecutionProfile** and the **SuperStepContext** are wrapped up in an **ExecuteStepPacket** which is sent to all **Slots** managing a graph partition. Once the packets are received on the **Slots** running on the worker nodes the **SuperStepExecutionProfile** is unpacked and the **Slot** loads the superstep implementation class through a class-loader.

4.6 Client API

For framework users sending commands to the framework through pure TCP/IP network links is complicated and error prone. Developers would need to understand the exact communication protocols. To provide simple means of client to framework communication a Client API was created. This client API from a pure technical perspective is using Java Remote Method Invocation (RMI)¹⁰ which is the Java implementation for

¹⁰<http://www.oracle.com/technetwork/articles/javaee/index-jsp-136424.html>

Remote Procedure Calls (RPC). The use of RMI hides many aspects of network computing from the developers which usually makes development easier and code better to read. RMI works by providing a Java interface which has an implementation at the server side and a proxy representation on the client side. Apart from some minor restrictions, the client can use the proxy object like a regular local Java object, although method calls to the object are routed through the network to the real implementation on the server side.

In `DynamoGraph` the service interface is defined in `DynamoGraphService` which provides all available service methods allowing for data manipulation, scheduling of super-steps, and retrieval of monitoring data. For the interested reader a detailed listing of the provided API functions and their usage is documented in appendix A.1.2.

For a client to create a proxy instance of `DynamoGraphService` it needs to use a special factory class called `DynamoGraphFactory`. This factory is necessary to hide details about the cluster configuration from the user. `DynamoGraphFactory` uses standard ZooKeeper configuration information to connect to a ZooKeeper service. This is best-practice also with other services built using ZooKeeper. The factory then queries the content of `/Master` to determine the IP address and port of the master process. With this information an RMI service registry is created that is then used to retrieve a connected `DynamoGraphService` object.

Developers using the reference implementation are highly encouraged to group calls to the Client API in blocks. These blocks should always start with a call to the `DynamoGraphFactory`. This way failures (and specifically master node failures) can also be tolerated by client applications. As displayed in listing 4.2 line 1 the call `ZooKeeperServer.defaultConfiguration()` uses standard configuration files to create connection parameters for ZooKeeper which are used to create a new `DynamoGraphFactory`. Line 2 of the same listing shows how this factory can be used to get a proxy object of `DynamoGraphService`.

Listing 4.2: A client application getting access to the Client API

```

1 DynamoGraphFactory factory = new DynamoGraphFactory(
    ZooKeeperServer.defaultConfiguration());
2 DynamoGraphService service = factory.getService();

```

Once a client application retrieved a `DynamoGraphService` proxy it is able to interact with the cluster. The available models in general fall into three categories (1) data manipulation methods, (2) algorithm execution methods, and (3) monitoring functions.

A full list of the available methods and their description can be found in online documentation¹¹ or in appendix A.1.2.

Data manipulation methods provide functionality to create, query, and delete namespaces. Namespaces are denoted by their names which that need to be unique for the system. On top of a namespace vertex, and edge methods can be executed to add, change, remove, and query single vertices. This also includes functions to add, change, and remove edges. Listing 4.3 demonstrates how a client application creates a new namespace (line 3) and inserts a newly created vertex (line 5).

Listing 4.3: Adding a namespace and inserting a vertex

```

1 DynamoGraphFactory factory = new DynamoGraphFactory(
    ZooKeeperServer.defaultConfiguration());
2 DynamoGraphService service = factory.getService();
3 service.createNamespace("test", Resolution.Weeks);
4 Vertex v = new Vertex(Resolution.Weeks, service.nextId("
    test", new Date()));
5 service.addVertex("test", v);

```

The next class of methods are those for executing algorithms on top of the framework. Functions that start a superstep algorithm are asynchronous. This means that developers call a method to submit the job to the cluster. The method call returns a job id which can then be used to query status and results from the framework. Users can pass parameters to the algorithms by setting in algorithm global context which is called **SuperStepContext** in the reference implementation. Further, developers can specify a time frame restriction which instructs the framework to only consider data in this time frame.

In listing 4.4 a code snippet is shown that executes a page rank algorithm (part of the algorithms shipped with the reference implementation) on a cluster which contains a namespace called *web* (service creation is omitted). In line 1 of the listing a **SuperStepContext** is created which is used in line 2 to configure a non-default damping factor for PageRank. In line 3 the job is actually submitted to the cluster and a job id is generated. This job id is then used in line 4 to wait for the superstep to complete and in line 5 to retrieve the results of the execution.

Listing 4.4: Submitting a job to the cluster

```

1 SuperStepContext context = new SuperStepContext();
2 context.put(PageRankStep.CONTEXT_DAMPING_FACTOR, 0.8f);
3 long jobId = service.executeAlgorithm("web", "at....
    PageRankStep");

```

¹¹<https://dynamograph.net>

```
4 service.waitForCompletion(jobId);  
5 context = service.queryAlgorithmContext(jobId);
```

The client API provides many more methods (an exhaustive list is given in appendix A.1.2). A set of API calls can be used to submit user-provided code to the cluster. Dynamic code loading and the related API methods are explained in detail in the next section 4.7.

Further the API also provides methods to retrieve monitoring data from the master node. This is explained in greater detail in section 4.8.

4.7 Java Dynamic Code Loading

This section discusses the implementation of a dynamic code loading module for the Java Virtual Machine (JVM). This addresses the requirement of users being able to upload their payload code to the framework and execute it in the context of the framework.

Since this prototypical implementation is based on the Java programming language and its accompanying execution model which is based on the JVM dynamic code loading for Java systems was used. The JVM per se performs dynamic code loading through a mechanism called the class loader. Every JVM during startup is configured with a default class loader that is capable to load code from different sources generally configured as the Java class-path¹². Java developers can leverage the mechanisms in the JVM by asking the default class loader to load classes from third party resources and by implementing a custom class loader. The latter has a big advantage since it also allows to unload code from the JVM that is not in use anymore.

As an example will highlight later the DynamoGraph framework allows users to dynamically load code through an API call. Developers need to bundle their custom code in Java JAR files and can send the code to the cluster for deployment. The master node distributes the code to all the worker nodes and makes sure that the workers deploy the JAR file. The custom class loader (in the prototype implementation called *VolatileJarClassLoader*) is able to handle these custom JAR packages. Uploaded code gets registered with an unique identifier which is the fully qualified class name of the main algorithm class. This allows the framework to chose the correct class loader implementation to load code for a certain superstep implementation. To avoid collisions

¹²Java CLASSPATH: <https://docs.oracle.com/javase/tutorial/essential/environment/paths.html>

with other packages, code dependencies (i.e. referenced classes, interfaces and enumerations) can only be loaded through the same class loader as the main class or through the JVM default class loader.

The setup described also allows for code unloading. During algorithm execution (as described in section 3.4) the framework registers as a *code user* with the class loader that provides code for a certain algorithm and unregisters once execution finishes. This way the framework is able to determine when it is safe to unload code from the system. Developers are able to unload their code through a simple API call.

The following example illustrates this dynamic code loading mechanism in greater detail. Let us assume a developer created a super-step implementation with in a class with the fully qualified name `net.dynamograph.code.MySuperStep` and needs a running cluster to execute this code package against a certain graph. In a first step the developer will make sure to wrap up the code and all its dependencies into a JAR file, most likely through the use of a tool like Maven¹³. Let us assume the JAR file is then named `mysuperstep.jar`. The developer now needs to complete the following steps to execute the code within the context of the cluster:

- Upload the JAR file to the cluster through an API call
- Use an API call to instruct the cluster to execute the code
- Unload the code from the cluster

Listing 4.5: Loading, execution, and unloading of code through API calls

```
1 DynamoGraphService service; // assume connected to API
   endpoint
2 String namespace; // assume initialized with a valid
   namespace name
3 Timeframe timeframe; // assume initialized with a valid
   timeframe
4 File testFile = new File("mysuperstep.jar");
5 FileInputStream in = new FileInputStream(testFile);
6 ByteArrayOutputStream bout = new ByteArrayOutputStream();
7 byte[] buffer = new byte[4096];
8 while(in.available() > 0) {
9     int num = in.read(buffer);
10    bout.write(buffer, 0, num);
11 }
```

¹³Apache Maven Project: <https://maven.apache.org>

```
12 in.close();
13 bout.close();
14 SuperStepContext context = new SuperStepContext();
15 String clasz = "net.dynamograph.code.MySuperStep";
16 service.loadCode(clasz, bout.toByteArray());
17 long id = service.executeAlgorithm(namespace, clasz,
    context, timeframe);
18 if(service.waitForCompletion(id) == ExecutionProfileState.
    Completed) {
19     context = service.queryAlgorithmContext(id);
20 }else{
21     // step execution failed
22 }
23 service.unloadCode(clasz);
```

In source code listing 4.5 a minimalistic example implementation of code loading with the DynamoGraph API is shown. Details about the API are discussed later in chapter 5, however, lines 1 to 3 initialise parameters such as the timeframe and the namespace used for the actual algorithm execution. In lines 4 to 13 the JAR `mysuperstep.jar` is loaded to a Java byte buffer which is later used to send the code to the cluster. The API service call in line 16 instructs the framework to load the code and identifies `net.dynamograph.code.MySuperStep` as the main class of this superstep. Assuming that code loading was successful (no exception was thrown) in line 17 the algorithm is executed in the specified namespace. The API call returns a unique id which can be used to query status information about the algorithm from the framework. In the lines 18 to 22 this minimalistic client waits for the algorithm to run to completion and in the case of successful completion retrieves the superstep context which contains results. Finally in line 23 the code for `net.dynamograph.code.MySuperStep` is unloaded again from the framework.

While this mechanism has certain advantages it obviously comes with some downsides. Whenever code is loaded from a third party it could potentially contain harmful code or just be erroneous and thus have a bad impact on system stability. The current implementation for instance will not detain developers from writing to the file-system on the compute nodes or prevent network access in any way.

The first case can be tackled by two strategies: The Java ecosystem provides a mechanism called the security manager which allows fine grained configuration of access privileges. One is able to control access to important system resources such as networking, the file-system, etc. while not implemented in the current state of the project it would

be relatively simple to write policy files for third party algorithms and to enforce these policies through adding security manager calls to the *VolatileJarClassLoader*.

A second option would be some form of code signing. Such that developers need to use asymmetric encryption to create a signature for their code packages. During the loading process the framework through the usage of a certification authority would be able to identify the developer and allow or deny code loading. This pattern is already popularity implemented at diverse mobile application stores such as Apple's App Store and the Google Play Store.

In the second case where a developer uploads potentially harmful code by accident more complex countermeasures need to be found. One case that comes to ones mind is a situation where break or termination conditions in the user's code are erroneous. It could be as simple as flawed loop logic that leads to endless loops or on a more global scale an algorithm that never votes for halt. The first case needs to be addressed through a mechanism that sets upper boundaries on the execution time of a single step during vertex local computation. The framework needs to cancel algorithm execution if this boundary is violated. These timeout settings will be a parameter usually chosen by a system administrator.

On the other hand the number of algorithm iterations might run out of hand since an algorithm falsely never votes for halt. In this case the decision needs to be left to the developer. Depending on the algorithm very long but also very short execution trails were observed in the tests of diverse algorithms. For instance the computation of vertex degree for all vertices in the network can be performed in a single super-step. The calculation of PageRank however takes 52 iterations on average to reach a stable result [89]. The problem can be addressed from two ways. First of all through the API it is possible at any time during algorithm execution to terminate a run. If it is not clear to the developers yet how many iterations their algorithms typically it is possible to terminate a run if it seems to take an unusual long time. Further it is possible to specify a maximum number of super-steps to be executed during a run. If this boundary is reached the framework automatically cancels the run.

4.8 Monitoring

As already argued earlier systems that are designed to run in a cluster composition have special requirements on monitoring. Firstly these systems are harder to debug because it is impractical to use common means of debugging (step-debuggers) on many different

computers in parallel. An second, in production deployment scenarios machine load needs to be closely monitored to react upon resource bottlenecks.

Monitoring is not in the center of attention for this thesis. Thus a tool-chain to monitor DynamoGraph was outsourced as a master's thesis [113]. The reference implementation, however, provides extensive mechanisms to collect monitoring data, stores it in in-memory buffers, and provides a client API to retrieve this data in bulk.

For the collection of monitoring data so called sensors are used. This is reflected in the framework by the Java interface `Sensor`. A sensor needs to be capable of providing a label that names the sensor and has to implement the method `getValue`. `getValue` must be implemented in a way to not affect system state, it must not create any locks on monitored data-structures. Each compute node (worker and master) during startup creates an instance of the class `NodeMonitor` the node monitor can be used to register an arbitrary number of `Sensors`. The `NodeMonitor` runs its own monitoring thread which in a configurable interval queries all registered sensors for their current value. On worker nodes the results are sent via a monitoring packet to the master node, the master node directly stores the results to in-memory buffers. This buffer is organised as a map where the key describes the node type, its IP address and port such that the origin of sensor data is encoded in the data structure. Clients can request the whole map to analyse different load aspects of the system.

Due to the simplicity of the implemented interval based monitoring model other more advanced sensors, such as event based sensors were omitted from the implementation. On the other hand, for crucial tasks like debugging of algorithms and system evaluation, specialized service methods were implemented which allow an application programmer to retrieve part of the system state on-demand. These calls include an function to retrieve details on algorithm execution, containing fields describing current global context, the number of active vertices, active slots, pending messages and overall algorithm state. Further, for long running data-import jobs the current import progress with details (number of failed records, number of active workers, etc.) can be retrieved if the import job is not a streaming data import and thus supports the computation of progress.

Sensors currently implemented in the system are the following:

CPU Sensor: This sensor returns an array with the CPU load of all CPUs installed in the system.

Memory Sensor: This sensor returns numeric values for the free, used, and maximum memory of the system.

Algorithm Sensor: This sensor returns a list of all currently executing algorithms (supersteps), their current phase and step, and the number of active vertices in the algorithm.

Counting Sensor: A generic counting sensor is available to count arbitrary events happening on any of the nodes. This sensor is currently used by the reference implementation to count the number of network connections that were established to any of the service endpoints. Sudden raise in this metric is a clear indicator for network instabilities.

Connection Pool Sensor is a metric that provides information on the use of connection pools. The number of connections in a pool should be relatively constant unless nodes are added or removed from the system. The sensor also returns the number of locked (used) connections, if this number is constantly high this means that communication bottlenecks are occurring on the worker nodes.

Partition Sensor: This sensor returns the number of currently active partitions.

Slot Sensor: This sensor returns the number of running slots in the system.

Timestamp Sensor: Upon each `getValue` provides the current machine timestamp. This is just a helper to assign the time of sensor data reading to the data.

Through the mechanism described, new sensors can be added to the system any time without breaking compatibility to tools using the sensor data.

4.9 Persistence Backends

By design it is possible to run the DynamoGraph framework as a stand-alone analysis platform. In this stand-alone mode all data is kept in memory. Restarts of compute node or the complete cluster will lead to loss of this in-memory model eventually. In the real world implementation this mode is called *Transient-Mode* and can be configured through setting the configuration value `PERSISTENCE` to `None`. Users running their cluster in this mode will need to make sure that the dataset will fit into memory on the available machines and before any experiments datasets need to be uploaded to the framework through API calls. Given the size of the datasets addressed by the framework this is a very time consuming process, leading to a situation where the *Transient-Mode* is actually only used in test scenarios where the framework itself or new algorithms implemented in the framework are tested. As described later 5.2 the

framework can also run in a stand-alone mode where all components of the framework are executed in a sequential manner in a single JVM for debugging of algorithms. In this stand-alone mode *Transient-Mode* is the only option.

In order to allow storing to a backend storage service certain persistence mechanisms were introduced. The framework can be switched to *Persistence-Mode* by setting the configuration value `PERSISTENCE` to the name of an actual persistence implementation. As discussed another requirement for the framework was modularity. This means that the persistence backend is implemented as a module and currently two implementations are available. A file based storage backend and one based on a Key-Value-store. Which ever storage mode is used the in-memory model also used in *Transient-Mode* is used as a cache for the graph model. The storage backend is implemented through an Java interface named `ModelPersistor` storage capabilities thus can easily be extended by implementing this interface.

4.9.1 File Based Persistence Backend

For development purposes and to minimize configuration overhead a file based storage backend is available. In this mode a configuration value called `PERSISTENCE_FILE_PATH` specifies a directory which every compute node in the cluster uses to create a node local directory for data storage. In this directory for each name-space in the system a sub-folder is created in which a collection of JSON encoded [1] documents is stored. Each of these documents refers to a single vertex (temporal map) in the graph. The filenames of the documents reflect to the vertex IDs in the network which allows direct access to the data.

This implementation is obviously limited by the local disk storage assigned to the worker nodes and is performance wise bound to the IO speed of the compute nodes. In this mode node failure will potentially lead to data loss unless the recovery actions are able to restore from the backup copies stored on the backup partitions.

A further limitation is that the JSON notation is comparably slow to binary formats in reading and writing vertices. This restriction turned out to be no limiting factor during development scenarios and allows for off-line analysis of the vertices on disk.

4.9.2 Cassandra Persistence Backend

A second implementation was created using a Key/Value-store to be precise the current implementation works with the Cassandra¹⁴ only. Key/Value-stores in general scale very well horizontally and are usually designed to be used in distributed computing environments. For the first implementation Cassandra was chosen as Key/Value-store since it uses similar underlying technology (Java, ZooKeeper) and is designed to be run in a distributed manner. In Cassandra this is called a *ring*. A *ring* is a distributed computing setup where each node is not only contributing storage capacity to the system but is also a fully qualified API endpoint to clients. This means clients can talk to any node in the system to read and write data. The Cassandra *ring* depending on concrete configuration manages the distributed storage and commit strategies for backup copies of all records.

The properties described make Cassandra the perfect choice to run in conjunction with the DynamoGraph compute framework. It is possible to run a Cassandra node and a DynamoGraph node on every compute node in the system. The worker nodes in DynamoGraph that use the Cassandra client API thus have local network access to the Cassandra *ring* which (again depending on configuration) allows for full usage of Cassandra's read cache. This will lead to optimal performance since read and write operations can mostly be executed locally.

The DynamoGraph namespaces are directly mapped to Cassandra keyspaces. Thus for every namespace in DynamoGraph a keyspace in Cassandra exists. On top of Cassandra two data storage strategies were implemented. Cassandra provides the concept of a column store which allows to store structured data in the system similar to relational databases. The column store is however used as a pure Key/Value store where the vertex id is used as the key and one data column is used as the value. The value is either a human-readable JSON document or a Java serialised binary representation of vertices, depending on configuration.

4.10 Future Improvements

As the provided reference implementation is a proof-of-concept for scientific use, there are still many points that leave room for improvement especially if it were considered to use DynamoGraph in production grade environments. In this section some of the improvements planned during continued research in this area are discussed.

¹⁴<http://cassandra.apache.org>

Further node to node communication is currently implemented through an abstract service layer that uses TCP/IP as a communication channel and is able to talk different encoding protocols (Java, Kyro and FST serialisation, and JSON) this layer is using in-memory input and output buffers that under certain conditions (severe network lag of a worker) can overflow and worst case can lead to worker node failure. This could be omitted by using proven message queue systems often used in other cloud-based service and Big Data infrastructures. Examples for message queue systems are RabbitMQ¹⁵, ZeroMQ¹⁶, and any implementation of the Java Message Service¹⁷. These systems are capable of buffering a large number of messages in a distributed, and ordered fashion such that it is still guaranteed that messages arrive at the recipients end in the same order as they were sent.

The current reference implementation also poses the restrictions that super-steps are scheduled and processed by the **SuperStepManager** in sequential manner. One could imagine that supersteps can also be executed in parallel, such that every slot on every worker runs an individual thread for each superstep. This change, however, would require severe restructuring of the worker nodes which, in the current phase of the project, is not desirable. The question that still remains is whether or not performance gains can be achieved through executing supersteps in parallel.

4.11 Concluding Remarks

This chapter has elaborated on the extended requirements when putting the theoretical model discussed in chapter 3 into practice. To prove that the presented approach is feasible from a pure technical point of view a reference implementation in Java was created. This chapter discussed the technical challenges faced during implementation and elaborated on their solutions. Different options for configuring certain aspects of the systems were presented, and a first reflection on room for improvement was made. The chapter gave first insights how software developers are able to use this framework for distributed temporal graph computing.

As this chapter addressed mainly the inner workings of the reference implementation of DynamoGraph, the Client API was only briefly discussed. Concrete usage examples and the use of the framework from a developers perspective can be found in the next chapter 5.

¹⁵<https://www.rabbitmq.com>

¹⁶<http://zeromq.org>

¹⁷<http://docs.oracle.com/javaee/7/api/javax/jms/package-summary.html>

Chapter 5

Usage of the Distributed Processing Framework

In the following chapter a brief introduction on, how prospective users (developers) can make use of the platform for their projects, is given. It covers typical use-cases 5.1 on a highly abstract level, execution modes 5.2, and setup scenarios 5.4 of the platform. Starting with section 5.6 case studies are discussed. These studies show how data originating from social networks, online social networks, but also the Web can be processed on top of the presented framework.

5.1 Typical Use-Cases

In typical use-case scenarios *DynamoGraph* will be used as a backend processing framework completely transparent to end-users. Typical applications will connect to various graph data-sources. These data-sources can be of various nature such as sensors, devices, and databases. It is not necessary that the data stored in these sources is already of graph structure. More often than that data will be some other type of raw-data that can be interpreted as graph data (e-Mail databases, Bluetooth proximity records etc.). As *DynamoGraph* allows temporal analytics over data it can also be used for near real-time applications. Where data is continuously fed into the system and metrics are continuously computed over the most recent data (See also section 5.5 for details on data ingestion). As outlined in figure 5.1 the first phase of a *DynamoGraph* application is thus referred to as *sensing*.

Sensor data is then streamed to the *modeling* component for further processing. *modeling* still contains tasks which are outside the scope of *DynamoGraph*. Such as the monitoring of sensors to determine if the sensors are continuously providing data and are not defective. Data cleansing that filters out erroneous data from the data streams

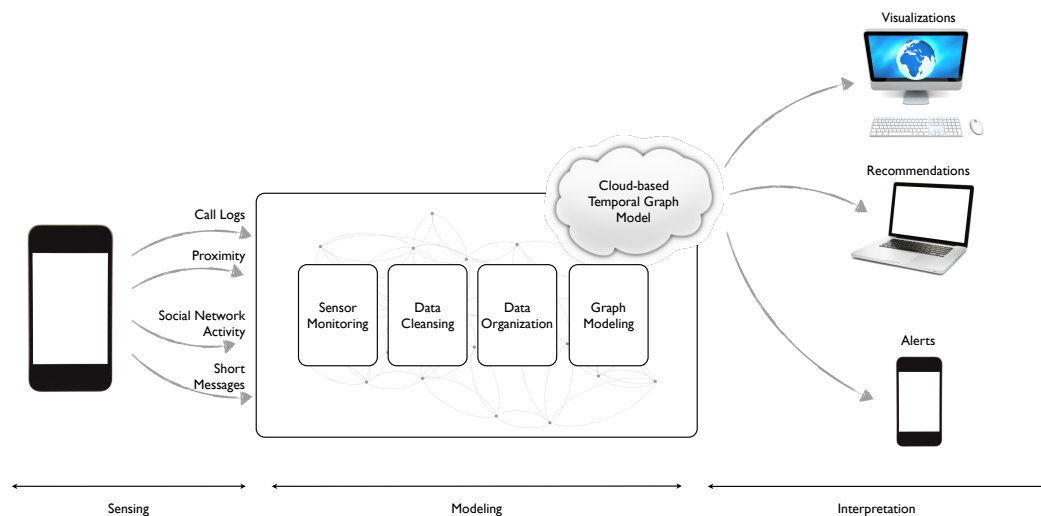


Figure 5.1: Typical architecture of an application using *DynamoGraph*

and omits records which are not useful for the task at hand. *DynamoGraph* then takes care for further steps in modeling such as the data organisation (i.e. partitioning data over many machines if necessary) and graph modeling (i.e. organising the data in named graph namespaces to allow user access) (see also the center block of figure 5.1).

Finally the data stored in the cloud-based temporal graph model can be interpreted for various applications. In this thesis the interpretation in a web-based visualisation is presented (sec 5.3). However, other applications might provide recommendations as results (see learning networks 5.6.1) or provide alerts in situations out of norm.

End-users will only touch *DynamoGraph*-based systems in their sensing and interpretation aspects. For instance a mobile application on a users smart-phone can provide sensing data and provide alerts.

5.2 Execution Modes

In general the platform is designed to address two very distinct application scenarios. Firstly, and quite obvious the platform can run in a production environment where data processing jobs are to be computed and the output of these jobs is the main concern. Secondly, the platform might also be run to implement compatible Pregel-style algorithms. In the latter case the implementation, and testing of newly invented algorithms is the main goal.

5.2.1 Cluster Mode

For production environments the processing framework will usually run on in *Cluster Mode* a compute cluster with the roles already discussed in section 4.2. It will usually be part of a larger software infrastructure such that users will not directly interface with it.

The actual implementation of *DynamoGraph* is a scientific prototype which leaves much room for configuration through the operators of the system. In general cluster mode runs a single instance of a JVM executing the master and worker roles on a single computer. As discussed in the architecture 4.2 each worker runs multiple parallel execution threads called **Slots**. Depending on the actual usage scenario and the available hardware configurations operators can decide to create almost arbitrary combinations of workers and slots. For instance for medium sized datasets such as the Enron e-Mail database single machines with multiple CPUs and sufficient large memory could be used. In this case operators will most likely run multiple slots on this single machine roughly running a single slot for each processor available to the system.

For large-scale datasets such as the IU Click Dataset which in its raw and uncompressed memory foot-print amounts for roughly $14TB$ large clusters need to be setup. The trade-off here is to dimension the cluster size in a way that as much of the graph can be fit into memory as possible to achieve optimal performance. For the raw dataset this will impose a memory requirement currently only found in high performance computing setups or with public cloud providers. Both very expensive options. Thus operators might decide to configure the cluster for persistent mode which means that data is actually stored in a Key-Value store and the in-memory representation is only a cache of this data. Naturally this will come at the cost of reduced performance.

5.2.2 Local Developer Mode

In the latter case, developers work with the system. They use it to develop algorithms and implement them as Pregel functions to be executed in the context of a compute cluster. However, many tasks that are easy in modern software development and thus have become the norm in the software development life cycle, such as automated testing, debugging, and memory inspection, become cumbersome in distributed software systems. To the developer a distributed system often seems like a black box where it is not immediately clear which computer is executing which parts of a system. Moreover in distributed processing paradigms like the presented a processing function gets executed on potentially many different computers in multiple threads.

Parts of the DynamoGraph framework can be executed in what is called the *Local Developer Mode*. This mode allows developers to keep small temporal graphs in memory on their developer machine and allows them to execute Pregel-styled jobs in a single thread in a single JVM over these local graph models. In local developer mode breakpoints can be set in jobs and developers can use step debugging and memory inspection to analyze the runtime behaviour of their code.

Further, automated tests (unit tests) can be and are implemented facilitating the local developer mode. Those tests are usually complex compared to tests performed in other software frameworks. They usually consist of steps for creating a temporal graph model in-memory, executing a certain job over the model, collecting the resulting metrics from the job, and asserting the results.

5.3 Web-based Graph Visualization

A still unsolved problem in large-scale temporal graph analysis is the visualisation of large graphs (see 1.2). Graph visualisation is a very natural way of making the information stored in a network accessible. Graph visualisation today is so pervasive that figures usually do not need further explanation and often case can be interpreted by laymen. If large-scale temporal graphs are concerned visualisation becomes far more difficult. In general there are two dimensions in the visualisation that are not adequately addressed today, the size and the time dimension.

The size of large scale graphs makes it inexpedient to use regular visualisation libraries and moreover automatic layout algorithms. Naturally traditional vertex and edge visualisations with millions of vertices are hard to visualise on screens that themselves provide resolutions of only a few million pixels. The result of such visualisations at best is what is known as a *hairball* (see example in figure 5.2 and if anything allows the viewer to compare the overall structure of networks with each other.

Further automated layouts for large-scale graphs become computationally very expensive. A popular method for graph layouting graphs is the use of force directed layouts. Their overall idea is that vertices (perhaps depending on some parameters such as the vertex degree) are subject to mutual repulsive forces. Such that no other counter-forces are applied they would drift apart. Counter-force is given due to the assumption that edges between vertices act as connecting spring bands. Iterative update of a graph model in a simulation converts the graph model from an irregular layout to a force directed layout. Already from the brief description given above it is clear that this category of algorithms has a worst case runtime complexity of $O(n^2)$ (n the number

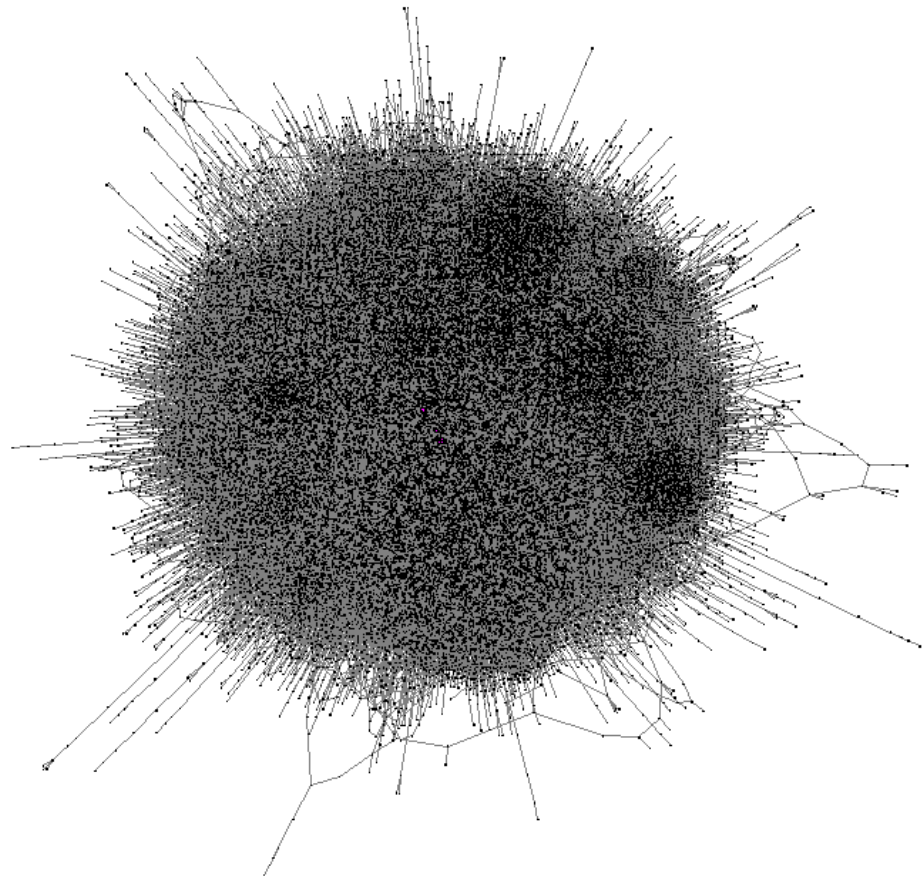


Figure 5.2: A hairball that provides perhaps a general overview but no details [83]

of vertices in the graph) because at least every vertex needs to be put in context with every other vertex of the graph on each iteration. With optimisations some algorithms can reuse computation results and achieve $O(n * \log(n))$ complexity [50]. Nevertheless, it is clear that complexity grows with the size of the graph.

The second problem with large scale temporal graph visualisation is adequate reflection of the temporal dimension. In general (see also figure 1.2 in section 1.3.1) for graphs of small size the visualisation of temporal graphs has been solved. Methods mainly use the 3D visualisation space to allow for the time dimension being added to the graph. A method that can be used is the 2.5D graph drawing method [46, 9]. One application of this method is to structure a graph into planes to visualise an otherwise hidden dimension. Although this method can also be used to visualise other structure such as clusters in the graph one main application is to show temporal snapshots of the same graph model thus allowing the user to observe changes in the graph. An example of such a model from an e-mail database is given in figure 5.3.

Although not on the core of this thesis also *DynamoGraph* provides basic means of graph visualisation. To make data most approachable to users a web-based visualisation

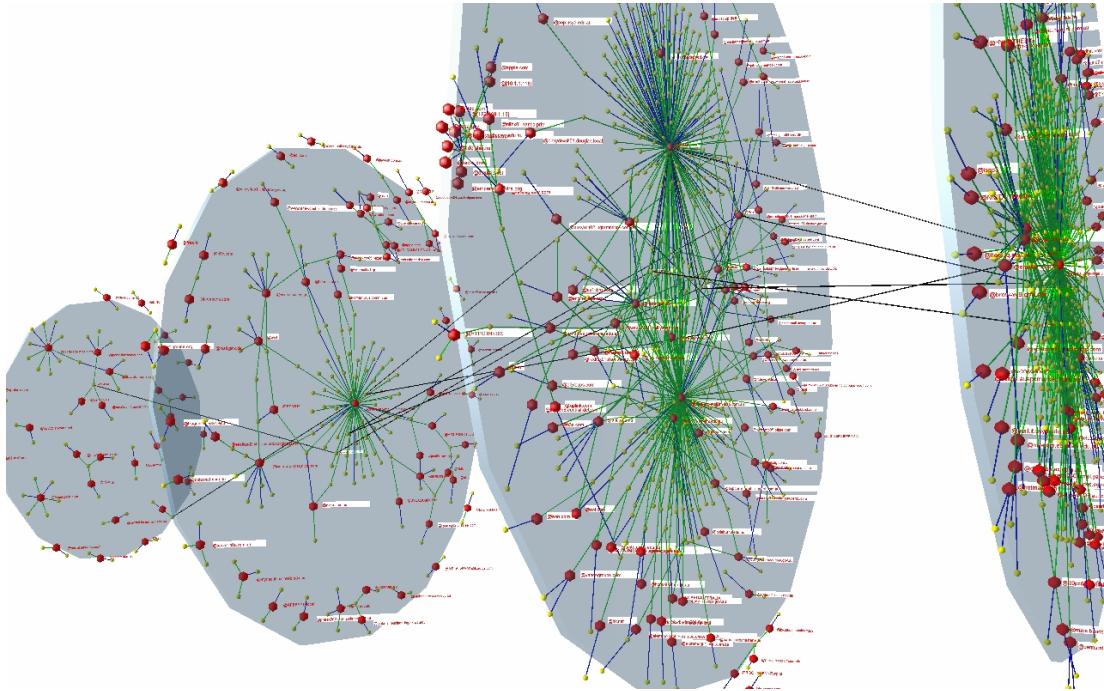


Figure 5.3: Temporal snapshots of an email database visualised using the 2.5D method [46]

approach was chosen. The project was implemented as a state-of-the-art Java Spring WebFlow-2¹ project. It enables users to access a running *DynamoGraph* cluster and provides tools for day to day data analysis tasks. This includes import of datasets (see also section 5.5), execution of distributed algorithms, and visualisation of graphs.

For visualisation the same restrictions described above hold such that also the *DynamoGraph* web-UI is able to displays graphs only up to a certain size. Administrators are able to configure this size during application setup. Currently graphs up to 200 to 500 vertices are still readable depending on the available screen sizes. For models larger (and even significantly larger) than that two approaches exist in the prototype.

Firstly, local neighborhood analysis can be used. In this mode general statistics (clusters, PageRank) are computed over the graph over the complete timespan available. From that certain interesting vertex-candidates to start neighborhood analysis are presented in a list. Such interesting vertices are the top-k ranked vertices according to PageRank or by vertex-degree. Alternatively users can directly search for vertices with certain attributes (vertex-id, name, etc.). From a selected starting point the neighborhood of the starting point can be loaded and visualised. Depending on the application scenario multiple levels of neighbors can be loaded (see the network query algorithm in section 3.8.3).

¹Spring WebFlow-2: <http://projects.spring.io/spring-webflow/>

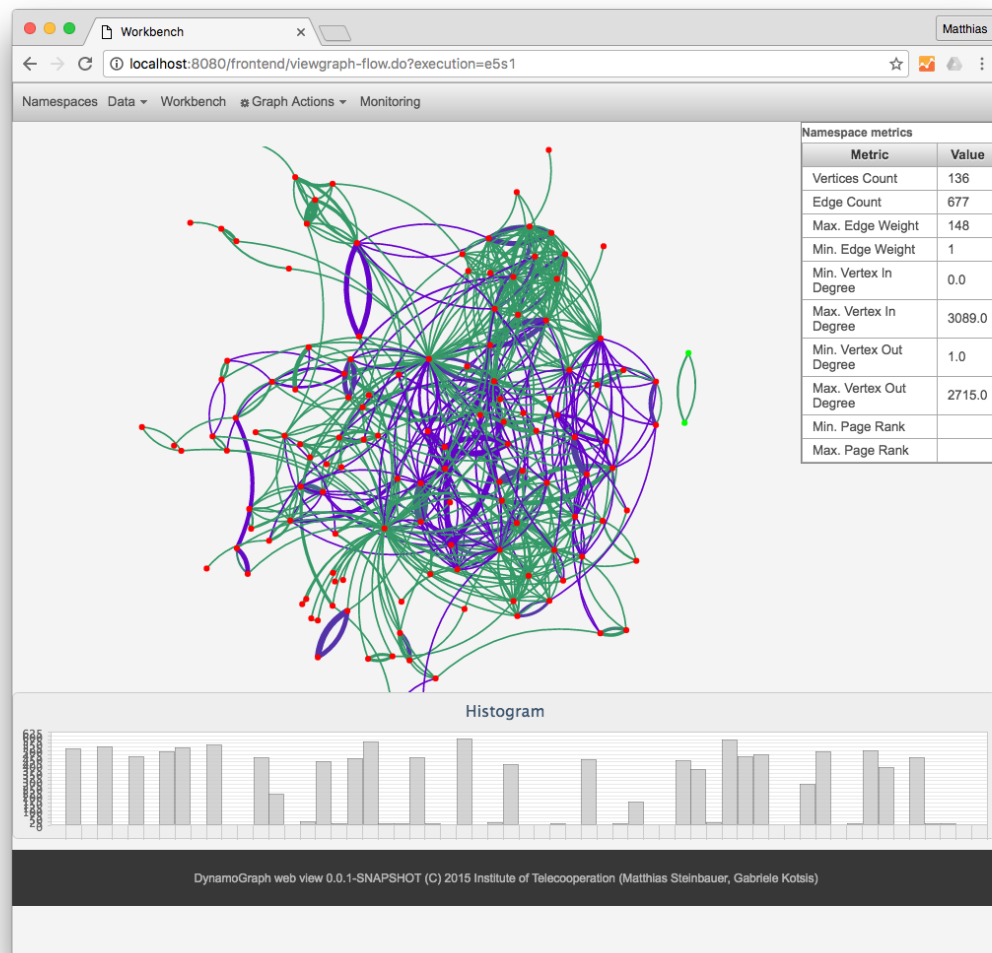


Figure 5.4: Calls and SMS from the MIT Reality Commons Network: Friends and Family [3]

Secondly, a fast clustering algorithm can be used to compute the network of clusters for the overall graph. For this the implementation discussed in section 3.7.4 is used. This allows the creation of several layers of cluster structure. Starting from the top layer the user is then able to dig deeper into the graph structure by navigating down layers on the graph stack. As this method is comparably complex software wise the implementation was outsourced as a supervised bachelors thesis titled *Layered Visualisation for Large-Scale Temporal Graphs in DynamoGraph* [92].

In any case *DynamoGraph* provides a histogram over the number of edges available in segments of time as a measure for users to find time-frames of higher or lower activity in the graph. Users can use the histogram to narrow the timeframe which is visible in the visualisation. An example of a network displayed in the *DynamoGraph* web-view is given in screenshot 5.4. It depicts a dataset from the MIT Reality Commons which contains edges of two types for phone calls (green) and SMS (purple) [3] (see appendix

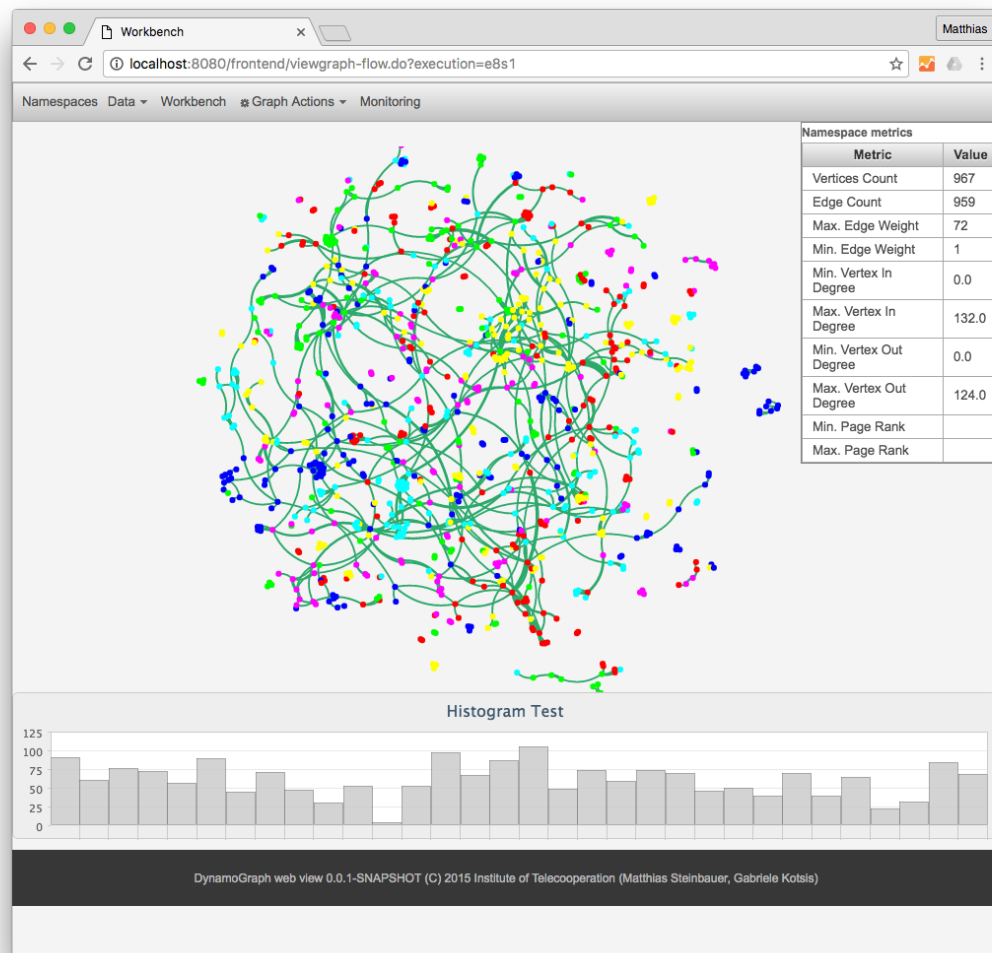


Figure 5.5: IRC Network with computed communities

B.1 for details on this dataset). The network was automatically layouted using the ForceAtlas2 [50] algorithm. At the bottom of the screenshot the edge histogram is displayed. It marks the frames of time with higher and lower activity. Some time-slots have no or almost no activity. This is due to the data collection process which was not done continuously.

Also in the screenshot basic information about the graph is displayed on the upper right corner. When a graph model is loaded by the user interface certain basic statistics are computed. One can also notice that the PageRank metrics are still blank in the screenshot. PageRank has a tendency for long runtime such that this metric is computed in background and the values become available only after the superstep run has completed on the cluster.

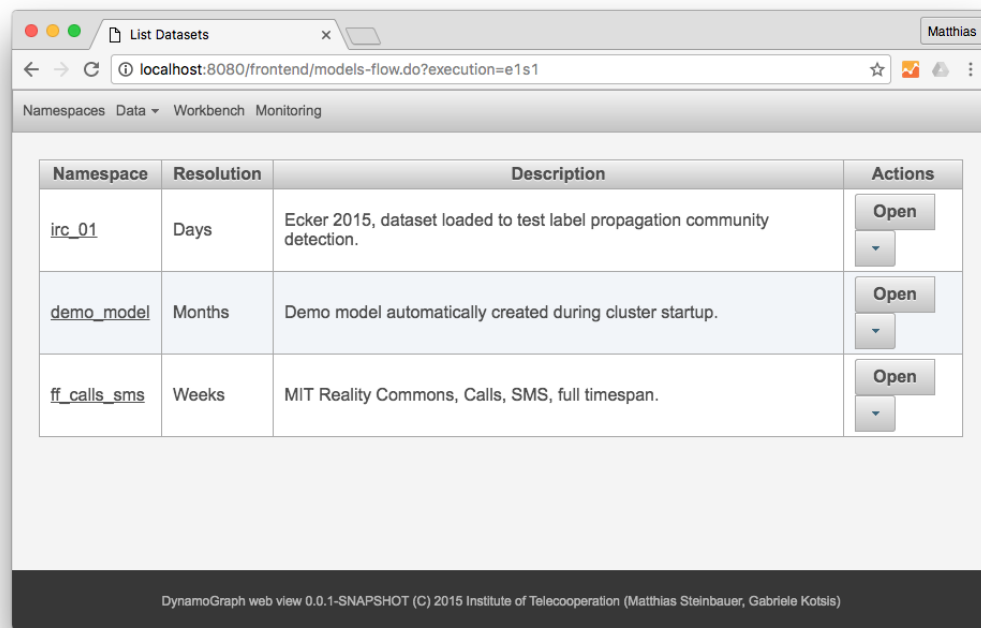


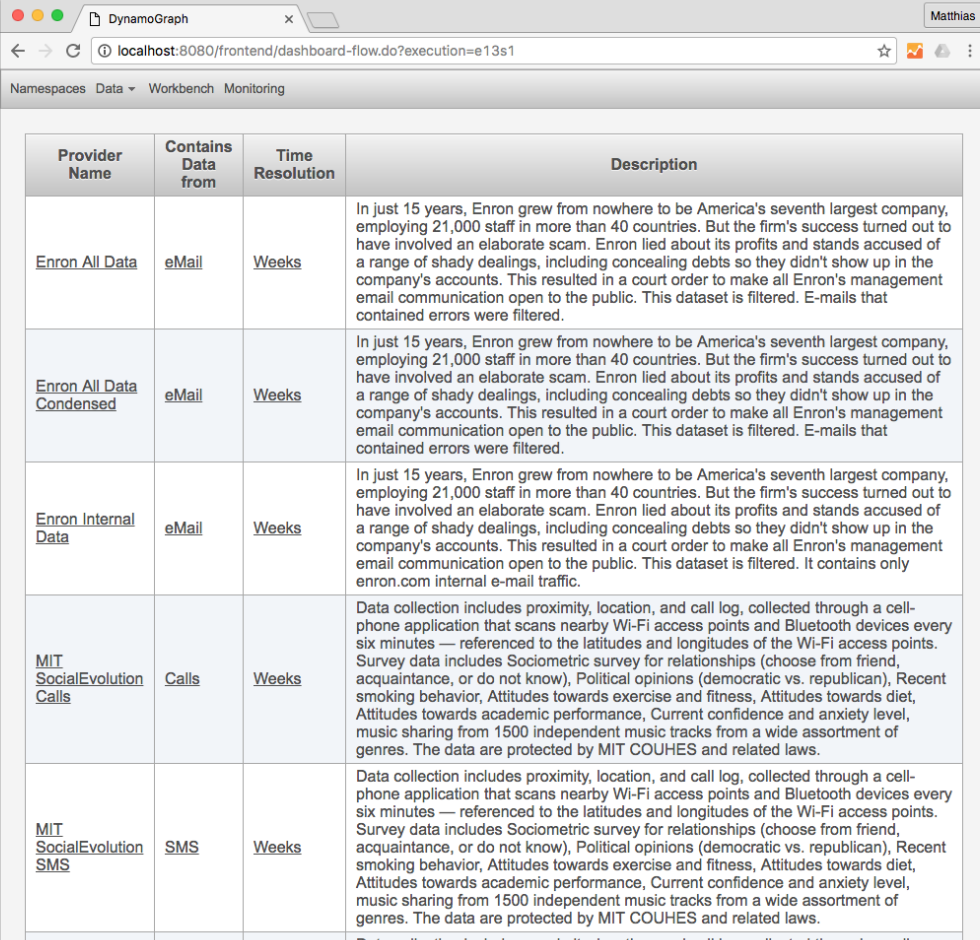
Figure 5.6: The namespaces view shows all datasets currently loaded on the cluster

A similar visualisation is presented in figure 5.5. It shows around 950 nodes from an IRC communication network [28], while the same visualisation technique was used a very different image was generated. Firstly, the vertices are colored in many different colors. Each color refers to a cluster / community and vertices of the same color belong to the same community (colors are repeated from a table). Secondly, comparably small edges got assigned an edge weight below 1 such that they are not visible in the visualisation anymore. This is configurable behavior and was turned on for this visualisation to avoid a hairball.

The user interface provides further options to work with a *DynamoGraph* cluster accessible through the top menu:

Namespaces allows the user to access a list of all graph models currently loaded on the cluster. The obvious operations are to load the model into the *workbench* view or to delete the model from the cluster. In this sense it gives a similar set of operations a user interface for a database management system would give for databases (see screenshot in figure 5.6). It serves as the navigation entry-point for users to access loaded datasets. It serves as the navigation entry-point for users to access loaded datasets.

Data / List data-sources can be used to load datasets onto the cluster. As explained in greater detail in the upcoming section 5.5 a sophisticated mechanisms to import static and dynamic data-sources into a *DynamoGraph* instance are supported. The *dataset*



The screenshot shows a web browser window titled 'DynamoGraph' with the URL 'localhost:8080/frontend/dashboard-flow.do?execution=e13s1'. The interface has tabs for 'Namespaces', 'Data', 'Workbench', and 'Monitoring'. The 'Data' tab is active, displaying a table of configured datasources.

Provider Name	Contains Data from	Time Resolution	Description
Enron All Data	eMail	Weeks	In just 15 years, Enron grew from nowhere to be America's seventh largest company, employing 21,000 staff in more than 40 countries. But the firm's success turned out to have involved an elaborate scam. Enron lied about its profits and stands accused of a range of shady dealings, including concealing debts so they didn't show up in the company's accounts. This resulted in a court order to make all Enron's management email communication open to the public. This dataset is filtered. E-mails that contained errors were filtered.
Enron All Data Condensed	eMail	Weeks	In just 15 years, Enron grew from nowhere to be America's seventh largest company, employing 21,000 staff in more than 40 countries. But the firm's success turned out to have involved an elaborate scam. Enron lied about its profits and stands accused of a range of shady dealings, including concealing debts so they didn't show up in the company's accounts. This resulted in a court order to make all Enron's management email communication open to the public. This dataset is filtered. E-mails that contained errors were filtered.
Enron Internal Data	eMail	Weeks	In just 15 years, Enron grew from nowhere to be America's seventh largest company, employing 21,000 staff in more than 40 countries. But the firm's success turned out to have involved an elaborate scam. Enron lied about its profits and stands accused of a range of shady dealings, including concealing debts so they didn't show up in the company's accounts. This resulted in a court order to make all Enron's management email communication open to the public. This dataset is filtered. It contains only enron.com internal e-mail traffic.
MIT SocialEvolution Calls	Calls	Weeks	Data collection includes proximity, location, and call log, collected through a cell-phone application that scans nearby Wi-Fi access points and Bluetooth devices every six minutes — referenced to the latitudes and longitudes of the Wi-Fi access points. Survey data includes Sociometric survey for relationships (choose from friend, acquaintance, or do not know), Political opinions (democratic vs. republican), Recent smoking behavior, Attitudes towards exercise and fitness, Attitudes towards diet, Attitudes towards academic performance, Current confidence and anxiety level, music sharing from 1500 independent music tracks from a wide assortment of genres. The data are protected by MIT COUHES and related laws.
MIT SocialEvolution SMS	SMS	Weeks	Data collection includes proximity, location, and call log, collected through a cell-phone application that scans nearby Wi-Fi access points and Bluetooth devices every six minutes — referenced to the latitudes and longitudes of the Wi-Fi access points. Survey data includes Sociometric survey for relationships (choose from friend, acquaintance, or do not know), Political opinions (democratic vs. republican), Recent smoking behavior, Attitudes towards exercise and fitness, Attitudes towards diet, Attitudes towards academic performance, Current confidence and anxiety level, music sharing from 1500 independent music tracks from a wide assortment of genres. The data are protected by MIT COUHES and related laws.

Figure 5.7: All currently configured datasources

view (see screenshot in figure 5.7) provides an overview over all such configured data-sources. Users are able to select and configure a data-source in order to upload it into a namespace on the cluster.

Monitoring finally provides information about the running cluster. Most interestingly there are general metrics about currently executing and previously executed algorithms. This view provides a quick overview over a clusters current processing status. In the log table users are able to click individual records to inspect the algorithm run in detail. For completed algorithms the detail view displays the contents of the `SuperStepContext` which refers to the global state Γ of the algorithm (see section 3.4.1). This mechanism can be used to report results of algorithm runs back to users. Further it can be used to execute arbitrary algorithms. As depicted in figure 5.8 a set of predefined supersteps can be selected.

Class Name	Name Space	Profile ID	Time	Start Time	End Time
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.StatisticsStep	irc_15	1471857091441	0	2016-08-22 09:11:31	2016-08-22 09:11:31
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.HistogramStep	irc_15	1471857091507	0	2016-08-22 09:11:31	2016-08-22 09:11:31
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.cluster.VertexLabelingStep	irc_15	1471857091530	15	2016-08-22 09:11:31	2016-08-22 09:11:31
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.query.QuerySuperStep	irc_15	1471857091786	0	2016-08-22 09:11:31	2016-08-22 09:11:31
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.StatisticsStep	irc_01	1471857126448	0	2016-08-22 09:12:06	2016-08-22 09:12:06
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.HistogramStep	irc_01	1471857126520	0	2016-08-22 09:12:06	2016-08-22 09:12:09
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.cluster.VertexLabelingStep	irc_01	1471857129296	15	2016-08-22 09:12:09	2016-08-22 09:12:13
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.query.QuerySuperStep	irc_01	1471857133318	0	2016-08-22 09:12:13	2016-08-22 09:12:13
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.query.QuerySuperStep	irc_01	1471857756610	0	2016-08-22 09:22:36	2016-08-22 09:22:36
at.jku.tk.steinbauer.dynamo.distributed.compute.steps.impl.query.QuerySuperStep	irc_01	1471857767448	0	2016-08-22 09:22:47	2016-08-22 09:22:47

Figure 5.8: Log of the last algorithms scheduled on the cluster

Together with the distributed cluster system the graphical user interface completes the *DynamoGraph* tool-chain. It is a first step towards making a system like the presented accessible to administrators, developers, and in far future perhaps also to end users. The complexity of the cluster setup is hidden from the user.

5.4 Setup and Configuration

As already discussed in greater detail in the previous sections and chapters there are two base components in *DynamoGraph* that need to be installed to form a working cluster: a single *master* node and at least one but possibly many *worker* nodes. The *master* node have shown only insignificant workload in tests such that the default provisioning model uses the *master* also as *worker* node.

A crucial component also required for all installations is a working instance of Apache ZooKeeper which is used for distributed configuration management and coordination. Additional optional components are a persistence backend and a graphical user interface. Depending on exact configuration persistence can require a working Apache Cassandra key-value store to be installed. The graphical user interface described in section 5.3 is a Java Servlet based web-application and thus requires a configured Servlet container for roll-out.

During tests and evaluation of *DynamoGraph* the software was setup in different configuration scenarios oftentimes. Such that for evaluation it became necessary to fully automate the software installation which is described in greater detail in the following. The automated roll-out is described in greater detail in section 6.3.3.

Most of the components of *DynamoGraph* are built in Java 8. Also the dependencies (ZooKeeper, Cassandra, etc.) are built on top of Java technology. This means that all hosts running any component of *DynamoGraph* require a Java 8 JRE installation.

5.4.1 Apache ZooKeeper

The central instance of configuration and coordination is an installation of Apache ZooKeeper. In the default case ZooKeeper can be installed alongside with the *master* and a single *worker* instance on a single machine. For ZooKeeper no extensive configuration requirements are given. *DynamoGraph* was developed and tested against ZooKeeper version 3.4.8 which can be downloaded in a vanilla distribution for Linux directly from the project website².

In order to run ZooKeeper on a Linux machine the vanilla release needs to be unpacked and a configuration file must be created. The configuration file requires the mandatory configuration attributes `ticktime` which denotes the heartbeat time in milliseconds, `dataDir` that specifies where ZooKeeper can store its persistent configuration database, and `clientPort`. The vanilla release of ZooKeeper provides a sample configuration file with useful defaults that can be safely used with *DynamoGraph*.

DynamoGraph usually have their ZooKeeper installation living inside of `/opt/-zookeeper` and automatically launch the instance on machine boot by executing `/opt/zookeeper/bin/zkServer.sh start`.

5.4.2 DynamoGraph Master and Workers

As soon as ZooKeeper has started a *master* and possibly multiple *workers* can be installed. Both software components are implemented in the *distributed* subproject of *DynamoGraph*. These components in general follow two paths of configuration. There is static configuration which is parsed and applied from a Java properties file and dynamic cluster configuration which is negotiated, stored and retrieved via ZooKeeper.

The properties file can be located in `/etc/dynamograph/global.properties` and in the users home `/.dynamograph/global.properties`. It contains general configuration settings which are meant to remain static during a cluster run. In practice the configuration file is parsed in regular time intervals to allow some of the configured attributes to be adapted during runtime. Typical settings found in

²Apache ZooKeeper Releases: <https://zookeeper.apache.org/releases.html>

`global.properties` are the number of *slots* (threads) to be preallocated for each worker (`SLOTS_PER_WORKER`), behaviour in case of node failure, optional configuration of persistence backends, and log settings for individual components.

Node failure behaviour describes whether or not backup partitions are to be created on the *worker* machines. If a failover scenario is configured failure of individual worker nodes or even the master can be compensated by the system to a certain extent. If a *worker* fails the **PartitionManager** component is able to redistribute the data stored in *DynamoGraph*, if the *master* fails the worker are able to elect a new *master* to allow for continuous use of the system.

The configuration file also contains the IP-address or hostname (`ZOOKEEPER_HOST`) and port of the ZooKeeper server. On application startup the application initializes according to the configuration file (slot, persistence backend, etc.) and then queries the ZooKeeper node `/Master` to retrieve the service endpoint of the *master*. If `/Master` is not available on ZooKeeper this means that the cluster has just started or that the *master* has failed. In either case a new election process is initiated (see section 4.3.1 for details). *Workers* register their own service endpoint under ZooKeeper node `/Workers/` which is a path that is monitored for change by the *master*. This way configuration changes in the cluster setup (addition or removal of *workers*) can be registered by the **PartitionManager** component and the *master* is able to react accordingly. In case a *distributed* component is started and no configuration file can be found a cluster with sensible default values (transient mode with 4 threads, assuming ZooKeeper on localhost with default port) is started.

In typical setup scenarios the *distributed* subproject is started on multiple computers in sequential order such that the first machine that launches the software component becomes *master* and all consecutively launched machines become *workers*. Depending on the configured cluster size and the time between the individual computers starting up and registering with ZooKeeper multiple reorganisation steps are executed. Such that it is advisable to allow the system to settle before uploading data to the cluster. The API allows to query the current cluster status such that even for a client observing the cluster no guesswork is involved in determining if all workers have properly started.

A binary assembly of the *distributed* project contains an executable JAR file and service scripts that allow administrators to register *DynamoGraph* as a Linux SysV service. It is strongly advised to use the init script since extensive Java JVM configuration parameters such as maximum memory and remote method invocation bindings are specified by the script. Without these settings the cluster will experience a severe performance impact. The service script can be copied to `/etc/init.d/dynamograph`

which allows for the service to automatically launch at system startup or to start it manually with `service dynamograph start`.

5.4.3 Optional Persistence Backends

In the *DynamoGraph* global configuration optional support for persistence backends can be specified (`PERSISTENCE`). Currently the options `None` for transient/volatile mode, `JSON_Files` for storing vertices in individual JSON files, and `KeyValueStore_Cassandra` to use an Apache Cassandra cluster are available. In the case of Cassandra it is advisable to run the instances of a Cassandra ring on the same machines as the *DynamoGraph* cluster. Cassandra supports extensive concepts for data locality and replication such that *worker* nodes can connect their local Cassandra instance and be sure that all vertices living in the *workers* partition are also stored on the local Cassandra partition. This avoids unnecessary network round-trips.

Cassandra clusters are organised in so called rings where data stored to a partition is automatically replicated to other partitions with configurable replication, locking, and commit behaviour. Due to this architecture each individual instance in a ring can be used as a server to clients. Since in a key-value persistence scenario it is assumed that the underlying storage framework is responsible for data replication, *DynamoGraph* is disabling failover features when running in `KeyValueStore_Cassandra` mode.

If non-standard behaviour for Cassandra is used (i.e. re-use of an already existing Cassandra cluster) then the Cassandra service endpoint can be specified in the *DynamoGraph* configuration (`PERSISTENCE_CASSANDRA_PORT`, `PERSISTENCE_CASSANDRA_HOST`).

5.4.4 Optional Web Interface

The optional web interface is acting as a regular *DynamoGraph* client. This means it connects to the cluster using the Client API. From an architectural point of view the web UI can run on any Servlet container and requires only network connectivity to the cluster. The application is standards compliant and as such can be configured via web-application contexts. The application itself provides a default context which can be overridden by the Servlet container. *DynamoGraph* was implemented and tested using the Tomcat 8 Servlet engine, which allows to provide external context in its installation path `$TOMCAT_HOME/conf/Catalina/localhost`. The web interface is implemented in the *frontend* project. This means a context configuration called `frontend.xml` can be created in the aforementioned path to override default configuration parameters.

In general there are two options to configure cluster connectivity. By either directly specifying an *DynamoGraph* API endpoint in giving the RMI host and port where a *DynamoGraph* master is running or in specifying a ZooKeeper configuration which can be used to automatically lookup the Client API endpoint in the ZooKeeper node */Master*. Obviously the second option is more robust in terms of a failing *master* component. However, in other scenarios, such as during development or if the service is made available through a public IP address, direct connection to a cluster might be preferable.

Listing 5.1: Example Frontend Context

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Context>
3   <Manager pathname="" />
4   <!-- ZooKeeper Configuration -->
5   <Parameter name="zookeeperHost" value="localhost" />
6   <Parameter name="zookeeperPort" value="2181" />
7   <Parameter name="useZookeeperDiscovery" value="true" />
8   <Parameter name="rmiHost" value="140.78.92.57" />
9   <Parameter name="rmiPort" value="1201" />
10 </Context>

```

In listing 5.1 an example context is given that shows both configuration variants. The parameter `useZookeeperDiscovery` can be used to switch between the two described behaviours and is on by default. Line 5 and 6 show the configuration from a test cluster where Tomcat and the web UI were installed alongside ZooKeeper and the *master* component on the first provisioned compute node in the cluster. Hence a ZooKeeper discovery of configuration can be performed on `localhost`. In lines 8 through 9 the direct configuration of our public *DynamoGraph* service used during evaluation is specified.

5.5 Data-set Import

Since the presented distributed computing platform and the web-based user interface discussed in previous sections are research prototypes it is an important requirement to quickly import, visualise and analyse data. Scientific datasets as explained in appendix B are often available as flat files for easy import to databases, and other data processing tools. The files are often formatted as CSV, XML, and JSON. Other data-sources are continuously sending new data from a live system (such as Twitter). Thus integrated with the web-based *DynamoGraph* platform an import utility was built.

The import tool is configured with a list of static and dynamic data sources. Users of the platform can choose any data source and import it to a namespace in DynamoGraph for processing. Static data sources are files that can be imported once. DynamoGraph currently supports CSV, and JSON import. An example for this would be the CSV files available from the Click Dataset [74] (described in detail in appendix B.3). Dynamic data sources are implemented as Java classes that continuously stream new data to the system. An example for this would be a service that retrieves Tweets with a certain hashtag live from Twitter and adds the information to the assigned namespace.

Administrators running the DynamoGraph platform are able to describe datasources in XML based configuration files. The system reads a file called *dataSources.xml* from the web containers resources path. This file contains a list of filenames of XML files that describe a single datasource in detail.

Each datasource description file is structured as follows. The XML root element is *data-Source* with the mandatory attributes *name*, *resolution*, and *type*. The *name* attribute denotes a human readable name that is displayed in the system for the user. *resolution* specifies the namespace that needs to be used for importing the data. Finally, *type* specifies whether the datasource is *Static* or *Dynamic*.

As a child element to *dataSource* a *description* element needs to be specified that provides textual description of the data provided by the data source. Again this information is being displayed in the DynamoGraph platform user-interface.

Further any number of *component* elements can be specified as children to *dataSource*. In the case of a static datasource the component denotes the datafile and instructions on how to read the data. In case of a dynamic datasource the component refers to a Java class name that implements the dynamic importer interface. A component has a mandatory *name* attributed and a *source* attribute which in the case of a static datasource denotes the file format used. Further the following XML child elements can be specified:

locator: The locator attribute describes how the system can access the datasource. In case of static file sources the locator can be a URL pointing to the file, or a path pointing to the file on disk. Paths can be prefixed by **res:** to denote that the path is relative to the application containers resources and points to a data-file that is shipped with the DynamoGraph installation.

colorCode: For visualisation purposes DynamoGraph supports the use of HTML color codes to color vertices and edges. Visualisation components in the web-based workbench read this attribute. In the case of multi-graphs the visualisation can

be configured to merge parallel arcs. In this case a color mixer automatically finds mixed colors for the merged arcs.

type: Each component has a string denoting the edge type of any arc that is resulting from said component. In the case of a multigraph data from different sources can be imported as parallel arcs. For instance a dataset containing a social graph from voice-calls and SMS can have parallel edges for calls and SMS.

weightCorrections: The weight correction is a float number in $]0, 1]$ and defaults to 1. In the case of a multigraph the weighting of edges of different type can be specified through weight corrections. To reuse the example of voice-calls and SMS one might assume that a voice-call causes a higher degree of social interaction than a SMS and thus might chose to weight voice-calls with 0.7 and SMS with 0.3.

configuration: Finally a text property configuration can be used to specify any other configuration parameters for the component. The text property is passed to the resource locator responsible for reading the data and is interpreted by the software component that performs the actual import process for the data component. In the case of CSV files the configuration property will describe the columns found in the CSV data and allows to specify a column delimiter. For JSON files the configuration property can be used to specify which attributes in the JSON document contain the temporal graph data.

An example of a static datasource is given in listing 5.2. It describes how a preprocessed subset of the data available in the Click Dataset [74] can be imported to the cluster. The file denotes that the preprocessed data is available in months resolution and is static and consists of a single component. The component can be imported using the CSV importer and the configuration property describes that column 0 and 1 contain the vertex ids which form the networks edge list. Through the property locator the import component is instructed to pull the listed CSV files through the HTTP protocol from the specified locations and import them one by one.

Listing 5.2: Example Configuration for a Static Data-Source

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <dataSource name="Indiana University Click Data HTTP Pull"
   resolution="Months" type="Static">
3   <component name="Clicks" source="CSV">
4     <configuration>source:0; target:1; sourceName:2;
       targetName:3; weight:4; timestamp:5; skipFirst:
       false;</configuration>
5   <locator>https://dynamograph.net/click/2006-09-out.csv,
       https://dynamograph.net/click/2006-10-out.csv,https

```

```

        ://dynamograph.net/click/2006-11-out.csv,https://
        dynamograph.net/click/2006-12-out.csv,https://
        dynamograph.net/click/2007-01-out.csv</locator>
6    <colorCode>#2BAA6B</colorCode>
7    <type>Click</type>
8  </component>
9  <description>
10    53.5 billion HTTP requests made by users at Indiana
        University
11  </description>
12 </dataSource>

```

In contrast in listing 5.3 the description for a dynamic datasource is presented. This datasource describes that Tweets on certain topics (hashtags) are to be received. The datasource is defined as to store data in hourly resolution and listens only for the hashtags Apple, Microsoft, and Google from the microblogging platform. The configuration property is intentionally left blank in this example. It will contain the account information of the Twitter API user (API key and password) which is used to login at Twitter.

Listing 5.3: Example Configuration for a Dynamic Data-Source

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <dataSource name="Twitter Hashtag Reader" resolution="Hours
   " type="Dynamic">
3   <component name="Tweets" source="net.dynamograph.twitter.
        TwitterComponent">
4     <configuration></configuration>
5     <locator>#Apple,#Microsoft,#Google</locator>
6     <colorCode>#2BAA6B</colorCode>
7     <type>Tweets</type>
8   </component>
9   <description>
10     All Tweets concerning the three tech giants.
11   </description>
12 </dataSource>

```

5.6 Case Studies

To demonstrate practical usability of *DynamoGraph* several experiments were conducted in real-world scenarios. Some of these experiments were case-studies with the purpose of showing that the framework can be used in a real-world application scenario. Other experiments were conducted under closer monitoring of the system to describe properties and behavior of the system in larger details. The latter are discussed in detail in chapter 6.

The case studies discussed in the following sections use *DynamoGraph* as a processing platform. They not necessarily compute over large-scale datasets but show that the framework can be integrated into larger software systems. Mostly the data is imported to the framework using the import mechanism discussed in section 5.5 and visualised by the web-based user interface (see 5.3).

Both case-studies are of scientific interest in two different communities. The first case study addresses learning network analytics in the context of modern learning platforms. Graphs originating from such systems will in the future scale to very large sizes. The second study addresses political networks as observed through Open Data interfaces. Also these networks observed on a global scale and in more detailed nuances can scale to very large sizes.

5.6.1 Global Social Learning Network Analytics

In the 21st century learning increasingly happens on the social web. Learning is evolving from an organized, class-room activity with clear roles for all participants (learners, instructors, etc.) and thus also a clear flow of information, to an unstructured interactive social process. In this process participants learn about topics of their interest, skills, and aptitudes in patterns suitable to their personal learning style, using a multitude of inhomogeneous systems.

All the actors involved in learning (learners, instructors, authors, etc.) and their interaction with each other or their interaction through stigmergy on learning artefacts forms a temporal multi-graph. Assuming one were able to record the global graph of all learning interactions this would result in the temporal data structure called the global social learning network.

Analytics on top of this data structure is interesting on two different scales: (1) On a local level the learning network and their temporal evolution of individuals can show an

individuals progress through different topics. This network is called the personal learning network (PLN) it gives insight into the skills and topics an individual is interested in, and allows to reason about how well a certain individual is interwoven with other individuals and topics. (2) On a global level however graph algorithms for can be used to put individual PLNs in context. For instance community detection algorithms can be used to determine the communities of topics found in the global learning network. The found communities can be used to suggest new learning paths.

In the theories subsumed as connectivism social interactions play the key role in the process of learning. Knowledge is acquired by a learner through interaction with her PLN. Interaction can and will often happen online, not only in a formal learning platform but also informally using all sorts of services found in the social web (Facebook Twitter, YouTube, Google Documents, Office 365, etc.). This leads to a situation where the data from these inhomogeneous systems becomes hard to collect and analyze.

This problem however can be overcome through the application of the Experience API (xAPI). It uses formal statements to describe learning interaction i.e. *actor X answered to question Y, asked by Z*. These statements are either generated directly by learning systems that support this standard (for instance Moodle), or are derived through scraping data from other systems such as Facebook groups.

The Institute of Telecooperation together with the partner Research Studios Austria FG designed an architecture to bridge the gap between xAPI statements and their representation as a temporal multi-graph on top of DynamoGraph. An overview of this architecture is shown in figure 5.9 and was also published in [39]. The current outcome of these efforts are research prototypes capable of importing xAPI data into DynamoGraph. This illustrates how an xAPI proxy component can be installed between xAPI statement generators, such as mobile applications, learning management systems, and scrapers that pull data from online social networks and a learning record store (LRS). The proxy operation is transparent for the learning process. This means that xAPI statements are forwarded without manipulation to the LRS. This means storage of learning artefacts, and learning interaction is still working as intended.

The proxy component however is capable of filtering the xAPI statements and is capable of translating user interaction into DynamoGraph API calls. Many of these proxy components can be used to support distributed homogeneous installations of learning systems and LRS. As DynamoGraph is scalable it can run in a cloud-based environment such that a large number of proxies can stream data to one central installation. This central installation refers to the global social learning network.

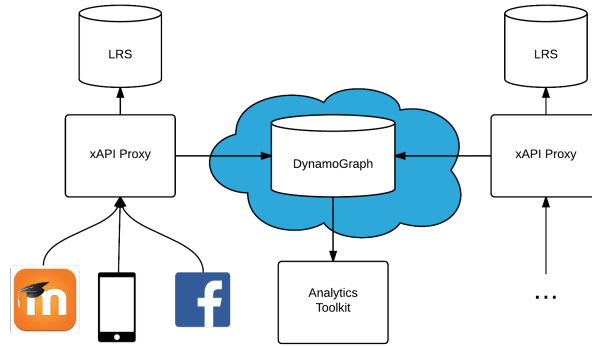


Figure 5.9: DynamoGraph in context with learning systems

Learning analytics toolkits can use functionality provided by DynamoGraph to visualize the data and to compute graph metrics on top of the network.

5.6.2 Temporal Political Network Analysis

A category of interesting temporal networks are political networks. These are formed on different levels of the political discourse and can provide a variety of exciting new insight. On a large scale online social networks such as Facebook [62] and Twitter [48] provide an interesting source for the automated analysis of political discourse.

If the lens is focused on a different point in the spectrum of political discussions then the discourse in official legislative bodies plays an important role. Especially in representative democracies the legislative bodies are the only instance directly responsible for policy making. Lately many initiatives make a move towards more transparent governmental structures which can be summarized as the Open Data movement. Open data can be coarsely categorized into political and social data, economic data, and operational and technical data [51] all of which are made available publicly to foster insight into otherwise in-transparent systems.

One of the oldest tools for increased transparency in parliamentary democracies are the transcripts of political debates. As means of documentation and later audit of discussion transcripts have been in place for many years. Usually these transcripts are created during debates by stenotypists, are then typeset and published as continuous volumes. Depending on country these transcripts have traditionally been made available to the general public through public libraries, and later with the advancements of technology through newspaper. As technology continued to advance today for many governments these transcripts are available for political professionals via e-mail subscription or as an Open Data service to a general audience.

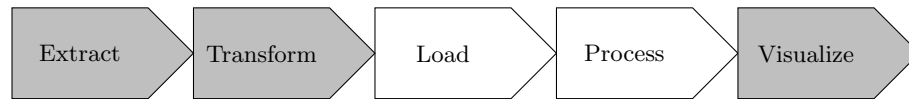


Figure 5.10: General structure of the processing pipeline

Although the availability of transcripts already creates a high degree of transparency for the democratic decision making process these still have some detriments. The general public will usually not be able to process all the transcripts simply due to the sheer volume of information making it too time consuming to process all the protocols. Direct analysis of the transcripts thus remains the task of professionals such as researchers, journalists and political analysts.

Further, simple statistics are hard to be made over the available text. Even professionals might find it difficult to keep track of the meta information of the political discussions, such as how often elected representatives engage in political discourse.

Finally, transcripts are pure text. From this text it is hard to derive the political structures inhibited in the political bodies. Even for professionals it might be sometimes hard to reveal the interweaving of policymakers in the network.

To make the data comprised in the transcripts of the Austrian parliament more accessible a case study with the tools provided by *DynamoGraph* was conducted and later published as a paper titled *Making Computers Understand Coalition and Opposition in Parliamentary Democracy* [106]. For this system a data processing pipeline was built that automatically pulls the HTML and PDF transcripts available as Open Data service³ and extracts all discussion blocks (see also figure 5.10). The discussion blocks of parliamentary discourse are annotated by the stenotypists of the Austrian parliament. The HTML transcripts already contain these annotations such that speakers and their general position in the discussion (pro or contra) is indicated in the data. These two steps are specific to this case study and are implemented without the support of *DynamoGraph* tools.

Further processing is then conducted inside of *DynamoGraph*. The data is loaded into a namespace and then in a processing phase a label propagation community detection algorithm and general metrics over the data are computed. To make the application possibly available for a general audience the visualisation phase is again implemented as a stand-alone web-application that is able to visualise the computed metrics. As shown in the screenshots in figure 5.11a and 5.11b these metrics can be general information such as the general presence or absence of clubs and individual politicians from debate in certain periods of time.

³Austrian Parliament Session Transcripts: <https://www.parlament.gv.at/PAKT/STPROT/>



Figure 5.11: Prototype screenshots

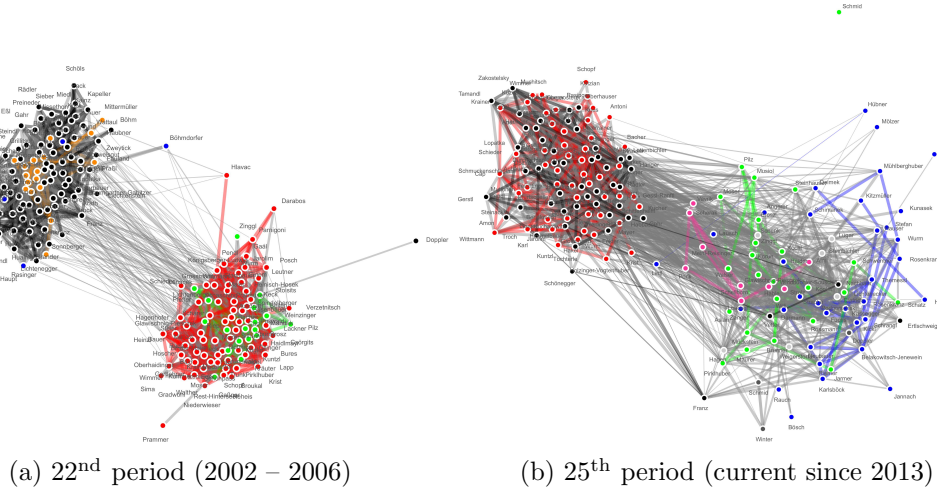


Figure 5.12: Politician relation graphs

However, more interesting is the outcome of community detection algorithm for certain time frames. In the figures 5.12a and 5.12b the 22nd and 25th period of Austrian parliament are given. The colors of the vertices in the graph refer to the official color of the political club the politician was assigned to at the beginning of the period. In the 22nd period the black, blue, and orange party formed a coalition after election. This is clearly visible from the force directed layout algorithm. Also the label propagation community detection could assign 98% of the politician profiles correctly to the coalition or opposition community. In the 25th period the red and black parties formed coalition and also this is visible in the automated layout and in the community labels assigned by the algorithm. Interestingly to observe is that the cohesion in the coalition and opposition clusters is quite different between these two periods. In period 25 coalition was formed by left and right-wing parties that naturally had difficulties to agree on common topics. This is also visible in the edges between the clusters that denote discussion statements with positive impact. There are far more of these links between the coalition an opposition cluster in 25 as compared to 22.

The presented case study shows that temporal graph data processing can be used in scenarios having direct impact on a large audience. Although the prototypical implementation of this research prototype is still in a very early stage its potential is already

clearly visible. This was also acknowledged by the eGovernment community as the paper was awarded with an outstanding paper award.

5.7 Prototype Summary

In the last sections it was shown that *DynamoGraph* provides all functionality required to serve as a research prototype. It was further discussed how this prototype was already used in case studies. This demonstrates that from a software-engineering point of view the technical feasibility of the approach is given. In the following more detailed evaluation is presented.

Chapter 6

Results and Discussion

The purpose of this section is to evaluate the presented computational model in the context of real world application workloads. This is to show the general technical feasibility of the approach, to provide assessment of the size of temporal graphs that can be addressed by the system, and to give insight of the scalability of the distributed computing framework. This is done by setting the evaluation objectives in section 6.1 and discussing how these objectives can be addressed by different evaluation methods (see section 6.2). The testbeds used during assessment are described in greater detail in section 6.3 and results conclude this chapter (see 6.4).

6.1 Evaluation Objectives

The purpose of this evaluation section is three-fold. First statements about the general *practical feasibility* are to be made. This work is application oriented science and as thus almost unintermediate use in real-world applications is a goal. As already discussed in prior chapters, the prototype was used in case-studies. This was done with the overall goal to survey how real-software development projects can use *DynamoGraph* and what features might be missing.

Secondly it is claimed that the approach provides *system scalability* as the problem size grows along network size and along the time dimension. Thus it must be shown that the system can handle workloads of various sizes in practicable manner. Further, some sense of scaling factors must be given as in which kinds of problem sizes require larger compute clusters. This way prospective framework users are able to predict on the computing resources needed for concrete cases.

Finally, *performance analysis* is required. It must be shown that the addition of compute resources allows to compute certain tasks in shorter time. Also for this evaluation

objective it is important that prospective users get a sense of real computing time and resources needed to address certain problem sizes.

6.2 Evaluation Methods

According to the evaluation objectives also the chosen methods are application oriented. The practical feasibility was studied using the discussed application prototypes and for the scalability and performance goals traditional real-world performance testing with system monitoring was used. The latter was done with real-world large-scale graph datasets for evaluation purposes after artificial graph models were used during development time to simulate workloads with saving time on the IO intensive data import phases. Datasets and methods are discussed in greater detail in the following sections 6.2.1 and 6.2.2.

Over the chosen datasets consistent but artificial workloads were run. The chosen algorithms were inspired by what is often found in related work. The workloads were executed over real-world hardware (see testbeds: 6.3) and resource utilization, log files and execution times were recorded for further evaluation.

6.2.1 Real-World Datasets

During the efforts of creating this thesis several datasets were used in order to prove its findings. Since the collection of a social dataset is itself a very time consuming process, this work mainly relies on already existing data to underpin findings.

Datasets available to research vary in size, quality of data, and the time span in which the data was recorded. Datasets that cover large-scale linked data from various sources are available from repositories such as the Stanford Large Network Dataset Collection [68] and the Koblenz Network Collection [61]. Both repositories focus on linked data and only a small number of temporal graph data sets.

In the domain of temporal graph data three data collections are to mention. First of all the Enron email database [20] which contains a large collection of e-mail communication from a real world company makes for a interesting object of study since e-mail databases of this nature can also be found in other real world application scenarios. For instance forensic investigators could analyse e-mail databases related to criminal cases or companies could analyse their communication structure.

Other datasets were created during scientific projects. One of these collections is the MIT Reality Commons which currently contains four datasets [27, 88, 3, 72]. These are very rich in terms of information that is contained in the data. All datasets were collected from a reality mining background where interaction between human beings was in the focus of research. The data recorded contains Bluetooth proximity information, GPS trajectories, e-mail communication, data about calls and SMS, and very often also data labels about activity and classification of relationships. From a size perspective the MIT Reality Commons dataset do not qualify as large-scale linked data. The graphs found in this data are usually smaller than 100 vertices.

A similar dataset can be found in the Nokia mobile data challenge [66] dataset which is also available to researchers. It is claimed to contain also information about calls, SMS, and Bluetooth proximity. However, this dataset could not be used since the process of acquiring access to this dataset is overly complex.

The previously discussed address either the temporal dimension of the large scale graphs. However, this thesis elaborates on the cross section of both. A dataset that can be classified as large-scale, temporal graph data is the Click Dataset (IU Click Dataset) available through the Center for Complex Networks and Systems Research (CNETS) at Indiana University Bloomington [75]. It is currently the largest web traffic network available to research. Compared to other large graph datasets they way the data was collected is well documented and it is clear that for certain types of HTTP traffic the dataset is also complete. This is in contrast with e.g. random walks on social networks which are used to crawl data. The quality of the data cannot be rated by outside observers.

For more details on the datasets used during evaluation, descriptions, structuring, and information on how to obtain them, the interested reader is referred to appendix B. For assessing scalability and performance parameters of *DynamoGraph* mainly the Enron email database and the IU Click Dataset were used since these are real world datasets of significant size.

6.2.2 Artificial Graph Models

In certain situations artificial graph models are a better option for testing graph processing systems. This is especially the case for situations where loading real world datasets takes up too much time and in situations where aspects of very large scale datasets are to be tested. As discussed in section 6.2.1 real world temporal graph data available for research is often too small to qualify as large-scale dataset (Big Data).

Artificial graph models formally describe the process of building a network. This formal description can be applied to an arbitrary number of vertices thus allows us to create networks of very large sizes. Artificial graph models feature parameters that can be used to control the process of graph building. Parameters for instance are the density of the graph and the probability that any two vertices are connected. In research artificial graphs are often used to test graph algorithms. Since certain parameters about the graph are known prior analysis these parameters must be visible as outcome of graph algorithms. The influence of certain parameters (e.g. the probability that any two vertices connect) can be studied in the context of metrics such as the average vertex degree and the graph diameter.

The parameters are designed in a way that they can be computed on naturally observed graphs. For instance the probability of any two vertices being connected can be determined in a given graph. This allows one to model artificial graphs closely after natural graphs such that artificial graphs resembling a social network can be generated.

The most famous model for artificial graph generation is the Erdős-Rényi random graph model [30]. The model describes a probabilistic process of selecting a random graph from the set of all possible graphs of a certain size. In practice this model can be implemented by initialising a graph of a defined number of vertices and using a random probabilistic function to define whether or not any pair of vertices in the graph are connected. This model can generate directed and non-directed graphs. A trivial implementation of this model can iterate over a defined set of vertices and choose a pair (or in the case of a directed graph an ordered pair) of vertices from the graph at random.

In a computer simulation it is trivial to use the Erdős-Rényi random graph model to generate an temporal graph model which for instance resembles a social network. This is done by monitoring the Erdős-Rényi process and assigning time labels on edge insertion. This will result in a temporal graph that behaves similar to a social network that communicates at a random pattern. The graphs produced by this generation model are different from real world graphs mainly in two aspects: (1) Since the probability of two vertices being connected is random and independent there is no local clustering as often observed in real-world networks. The resulting graphs have a low clustering coefficient. (2) Further in Erdős-Rényi no hubs are formed i.e. no nodes that act as brokers between clusters are found. Formally this refers to the fact that Erdős-Rényi graphs have a vertex degree distribution which converges to a Poisson distribution rather than to a power-law as observed in many real-world graphs. This means that an artificial temporal network generated according to Erdős-Rényi cannot be used to analyse the outcome of temporal graph algorithms. It can however be used to reflect

on the runtime behavior of graph algorithms and the scalability of graph processing systems.

In real world networks a property called the *small-world* property is most often found. As described by Watts-Strogatz one can label a graph small-world [121] if short average path lengths and high clustering coefficients are observed. The Watts-Strogatz model argues that there are random graph models for graphs with out randomness (regular graphs) and graphs with maximum randomness (i.e. Erdős-Rényi random graphs). However, real world graphs are somewhere in-between. A random graph construction is proposed where the starting point is a regular ring lattice of size N with each vertex connected to K neighbors on average. The algorithm constructs this initial network in a way that the maximum distance of vertices is limited. Assuming that the vertices are labeled v_1, v_2, \dots, v_n this means that vertices are only connected if the vertices index $i < j$ and $|i - j|$ is below a configured boundary. This results in a graph with strong local clustering. The process then continues to iterate through all edges of all vertices and rewires with a certain probability β under the condition of avoiding self-loops and link duplication. The rewiring process introduces randomness in the graph and creates so called short-cuts which later make for the small-world property of the graph. By conducting an endless number of rewiring iterations the graph converges towards an Erdős-Rényi random graph.

From the process described by Watts-Strogatz one can deduct that the graphs will closer resemble graphs found in nature. However, the process makes it more difficult to create a temporal graph. Iterations of the process will not add to the graph or reflect to the temporal development of the graph but will only introduce more nuances of randomness.

The problems with Erdős-Rényi random graphs and the Watts-Strogatz graph model where later picked up by the Barabáss-Albert preferential attachment model [5]. In their work it was discovered that neither of the aforementioned methods cover the evolvement of networks accurately. In turn a model is proposed that relies on the assumption that real world networks possess a degree distribution that follows the power law. Barabáss' and Albert's central assertion is that scale-free networks can be constructed what is called preferential attachment. This means that edge added to a network are more likely to connect to vertices with higher vertex degree. The model assumes that the probability that a vertex V attaches to vertex W is proportional to the degree of V and W . The process is covered in very many details and covers aspects such as different non-linear models of preferential attachment as found in natural networks and the initial attractiveness i.e. the probability that an isolated vertex gets connected to the graph.

In the context of large-scale temporal graphs the Barabási-Albert preferential attachment model again describes an iterative process which can also be used to construct a temporal graph. During the process of inserting new edges (or arcs) time and date labels need to be attached at random or following a certain pattern.

In practice for the models presented many different implementations exist. In order to use them for experiments one can rely on readily available software packets. Two software packets important to mention are the graph generator functions [84] that come with NetworkX a graph analysis system implemented in Python. The project supports generation of many different types of graphs.

6.3 Testbeds

The prototype was tested in cloud based environments. During evaluation the cloud stacks described in the following were used. One can easily roll out the software on other cloud stacks as long as Java and the possibility for private networks is available. In section 6.3.1 an private cloud installation available at the Institute of Telecooperation at the Johannes Kepler University Linz is described. Section 6.3.2 reflects on the learnings from using the Amazon AWS public cloud.

It is important to note that the evaluation results always need to be seen in context of the used cloud system. For private cloud installations one is able to observe the system in much greater detail since the physical hardware running the cloud stack can be observed and it is possible to understand the workloads of other users using the same system. This is not the case for public cloud installations. In general a public cloud provider offers only very high level meta-data about the virtual machines and physical roll-out of these machines in the cloud stack. This means one has no means whatsoever to understand the workloads of other users on the same hardware. On the other hand public cloud installations offer the possibility to run very large setups of cluster systems since the cloud providers allow one to scale the system on demand to clusters of hundreds or even thousands of machines which is not feasible in an on-premise approach. Consequently during the evaluation of *DynamoGraph* the private cloud installation was used to demonstrate the scalability of algorithms written in the framework and to analyse performance. The public cloud was used to demonstrate that the software provided in the *DynamoGraph* project is of high practical relevance and can be put to immediate use in real-world software projects.

6.3.1 OpenStack Private Cloud

The OpenStack¹ based private cloud installation available at the Institute of Telecooperation (see A.2 for exact software versions) was specifically purchased and designed to run research projects in the cloud computing and big data domain. Thus no software tools in production state run on the system and researchers are able to reserve the system exclusively for a certain amount of time which was made use of during evaluating DynamoGraph.

The available cloud-stack is comprised of three servers and a network-attached storage system. One server is acting as the front-end node for the system and runs the cloud-stack management software. It is the end-point for users to roll-out virtual machines on the system. The remaining servers act as compute nodes and are both equipped with two 6-core Intel(R) Xeon(R) X5690 CPUs running at a clock rate of $3.47GHz$ and $144GB$ of random access memory. This sums up to 24 physical CPUs and $288GB$ of RAM available for experiments. In the configuration of the system it is made sure that virtual machine disk images currently used by the system are stored on local storage, the network-attached storage system is used for backup and storage of datasets only.

All the machines are running the open source operating system CentOS 7. The virtual machines are run on top of the Linux standard KVM² (Kernel-based Virtual Machine) hypervisor. KVM is widely adopted by practitioners to run virtualised software stacks. Further, KVM is the default hypervisor for OpenStack which provides in depth support for many of the features provided by KVM (memory ballooning, CPU over-provisioning, etc.).

Networking in the described cloud stack is implemented through the software defined networking service OpenVSwitch³ which is also the default option for networking in OpenStack. OpenVSwitch is a multilayer open source virtual switch. It operates on layer 2 of the OSI network model and acts as and integrates with regular self learning layer 2 switches. This means that OpenVSwitch can be used to emulate Ethernet network segments. In the context of cloud computing it becomes possible that individual projects and software roll-outs are assigned their own private, switched Ethernet network. This minimises the risk of interference between individual software installations on the cloud, given the underlying physical network layer provides enough bandwidth to accommodate the workload of the virtual networks.

The network stack configuration used here creates some peculiarities that also become visible in the evaluation results. OpenVSwitch links which resulted in virtual machines

¹OpenStack: <https://www.openstack.org>

²KVM (Kernel-based Virtual Machine): www.linux-kvm.org

³OpenVSwitch: <http://openvswitch.org>

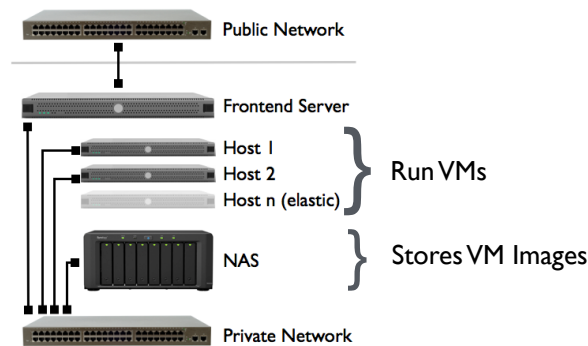


Figure 6.1: Schematic of a typical IaaS cloud installation

communicating on the same bare metal host do not need to travel through a physical network. Network traffic can be passed between virtual machines in-memory. This leads to network speeds well beyond 10 gbps. In the cloud installation a dedicated hardware switch was used to interconnect the bare metal hosts. This switch is an enterprise grade 1 gbps switch. This makes virtual machine to virtual machine traffic at least in the order of 10 times faster if both machines run on the same bare metal as opposed to real network links are utilized.

OpenStack provides a REST based API which can be used by third party applications to provision servers, networks, and storage. This allows to fully automate the process of setting up infrastructure, and installing software and running software. The OpenStack API is in wide areas compatible with the Amazon AWS EC2 API which makes it easier for infrastructure stacks designed for OpenStack to be migrated to the Amazon AWS Public cloud.

6.3.2 Amazon AWS Public Cloud

To allow tests under conditions one would find in a real world public cloud scenarios, setup and roll-out routines for the Amazon AWS Public Cloud were made. The offerings from Amazon were used since they are widely used by practitioners, their system provides reasonable private network isolation, most of the API endpoints are compatible with the OpenStack API and most importantly because Amazon provides grants for research and teaching⁴.

⁴Amazon AWS Research Grants: <https://aws.amazon.com/grants/>

Amazon AWS provides data-centers around the globe and allows its customers to dynamically provision compute resources in any of the provided locations. The AWS cloud further features a product called the Virtual Private Network, which is a software defined network stack that allows AWS users to create virtual layer 2 Ethernet networks that even span multiple of Amazon’s data-centers.

It is known that the hypervisor which powers AWS is based on the Linux Xen hypervisor. In general AWS provides a very similar feature set compared to OpenStack. Such that creating cloud-based software compatible with these two stacks creates only minimal overhead. The application roll-out described in the following thus has been used for scalability and performance testing on top of the OpenStack private cloud, and for technical feasibility testing on top of Amazon AWS.

Due to commercial and time restrictions and the upfront investment into the private cloud installation already made, no performance and scalability tests were conducted on top of the public cloud stacks. Especially the evaluation with the larger dataset would have cost of around \$2000, — per month as estimated by calculator tools provided by Amazon AWS.

6.3.3 Application Roll-out

In order to roll out the experiment installations on top of OpenStack and Amazon AWS a cloud agnostic middle-ware system was used which was also developed at the Institute of Telecooperation [107]. This allowed to fully automate the process of running evaluation runs which usually consisted of provisioning servers and networks in the cloud, rolling out a cluster installation, uploading a dataset, running the software, monitoring the system, collecting the results and tearing down the provisioned infrastructure. In order to avoid unexpected results from possible caching and optimisation by the used operating systems and cloud-stacks all used virtual machine instances used in an experiment run were created from scratch and deleted after each run.

On top of the middleware and the APIs provided by the cloud stacks a set of provisioning and tear-down scripts were created. Provisioning is done in four phases (1) network provisioning, (2) server provisioning, (3) software and configuration management, and (4) software execution.

Network provisioning: Cloud stacks provide software defined networks (SDN) which allow users to describe complex network infrastructures completely in software. The virtual network infrastructure provided by the cloud stack is by default decoupled from the physical transport network and topology. Servers running inside of

the cloud stack must explicitly be connected to physical networks through bridge ports that bridge network segments from the software defined network into the physical network.

For in the used provisioning scripts for each *DynamoGraph* cluster an separate private network segment was created in top of the SDN. In the used cloud stacks private network segments are completely isolated from all other network segments unless virtual routers are created that route traffic between network segments. This network is later used to connect the master and worker nodes of the cluster. The provisioning script also connects this private network segment to the public Internet with a router capable of network address translation (NAT). This allows the servers on this network segment to retrieve data from the public network (i.e. software packages, data-sets) but the servers cannot be reached from the outside.

Software defined networks also provide mechanisms for firewall configuration and advanced network configuration. Firewalls on routers are setup such that the nodes in the cluster are allowed to use the ports necessary for a *DynamoGraph* cluster. Further public IP addresses and host-names are obtained from a pool of available IP addresses and the user-facing services of *DynamoGraph* (mainly the API and the web-based user-interface) are made available to the public Internet through a public IP address and TCP/IP ports being forwarded to the appropriate virtual servers.

Infrastructure provisioning: Once the network is setup servers according to the experiments size are provisioned through a script. The script uses already pre-configured operating system images (in this case CentOS 7⁵ with extensions for cloud stacks and configuration management). It is first checked whether a current version of the operating system is available on the cloud stack. If this is not the case the image is obtained from the operating system provider and uploaded to the cloud.

With the operating system image available in the stack the script proceeds to provision servers. The provisioning software allows parameters for the number of worker nodes and their exact configuration (number of processors, memory, etc.) to be specified. Configuration was varied in different evaluation runs. Once servers are provisioned they are booted by the cloud stack, during boot apply generic operating system configuration as supplied by the stack (network configuration, hostname, user and authentication configuration, disk partitioning, etc.). After that the servers are in running state, users can login to the machines through the cloud stacks management interface or through secure shell (SSH) connections established through the private network segment.

⁵CentOS Project: <https://www.centos.org>

Software and configuration management: The operating system image used for *DynamoGraph* is prepared for configuration management by the Puppet⁶ IT automation software. All software and related configuration parameters running inside of the virtual servers is managed by Puppet. Puppet uses a descriptive language to describe desired system states. This system state can contain rules for certain software packages to be installed, firewall rules to be applied, services to run on system boot etc. Multiple rules can be compiled together to describe the desired state of a single computer or groups of computers. These sets of rules are called manifests. The Puppet software (Puppet agent) running on a server parses manifests and establishes the described system state.

Directly after virtual servers have booted they read the *DynamoGraph* manifest. Puppet installs all software dependencies such as ZooKeeper (on the master), Java, and Cassandra. Then the software packages of *DynamoGraph* are copied to their corresponding locations and configuration files are written.

Software execution: After all virtual servers have completed infrastructure provisioning and subsequently the software and configuration management phase, the script proceeds to run the cluster software. Due to the design of *DynamoGraph*, which allows for automatic election of a master role node, all servers execute the same *DynamoGraph* software package. Each instance of the software connects to ZooKeeper to retrieve configuration information and for master election. The server running the web-based user interface is started only after a master server was determined.

6.4 Results

At the end of chapter 5 two application scenarios for *DynamoGraph* were already underpinned with concrete case studies. While these case studies not necessarily fall into the category of very large-scale graphs they still illustrate how the system can serve as a platform for temporal graph analytics. The cases were selected in such a way that the underlying data-structures are subjective to growth. For instance in the analytics of political networks instead of the network of a single legislative body the global political network of multiple such instances can be analysed.

The case studies show that the mere functional requirements are well supported by the current prototype implementation. Individual instances could get configured as per application scenario and the software developers of the case studies could rely on

⁶Puppet: <https://puppetlabs.com>

DynamoGraph as an application framework. The requirements of the case studies were implemented without the need to adapt code in the framework. During the extensive development process multiple facets of the system were under test: configurability, multi-tenancy, modularity, persistence and dynamic code loading requirements have shown to be feasible in practical application scenarios.

Configurability was used in all case studies, for instance data granularity and persistence modes were tested in various setups. Usually prototype development for data processing pipelines was conducted using the transient mode where no persistence backend is used. In later stages of the project the Cassandra backend was used to permanently store data.

The *multi-tenancy* requirement was tested in initial phases of the case-studies due to the size of the datasets used it had quickly proven that separate, local *DynamoGraph* installations allowed for a simpler development process. In general for data management the multi-tenancy approach in the system is feasible, however, current superstep execution is implemented as a batch processing queue which means that the cluster processes supersteps in their order of submission. Later jobs stall until earlier jobs complete. This behavior will become a problem in production environments because a tenant would be able to lock a cluster infinitely.

In the case of *modularity* it is clear that the componentized architecture of *DynamoGraph* integrates well with other software architectures. For instance in the case-studies not all components of the system were used. Mostly the optional web-frontend was only used during development phases and also persistence backends were only configured as required.

For the *persistence* scenario it was shown that the backend is capable of storing and retrieving graph data. In general the datastructures composing vertices are temporal maps which allow the application developer to store additional attributes to vertices. This is used for instance for community detection algorithms where the community labels are written back to the temporal map for later retrieval. What the current implementation is missing, however, are more powerful query mechanics which would allow a developer to manually inspect the data. For the file based backend, although human-readable JSON files are stored, already starting with two partitions it starts to be complicated as to where files for certain vertices are stored. The Cassandra implementation makes use of serialization features to store non-standard attributes which makes them non-human-readable when inspected through Cassandra data access tools.

Finally, the *dynamic code loading* feature has proven to be very practical in real world scenarios. In typical use cases algorithms were implemented and their parameters adjusted using the local developer mode where a single executor thread is running the jobs in a serial controlled manner. The compiled and tuned algorithms were wrapped up in JAR files and submitted to the cluster in order to execute real payloads. Submission, registration, execution and de-registration of algorithm packages worked seamlessly in most cases. Of course also *DynamoGraph* hits boundaries common to the JVM, such that application developers need to make sure that no object references to side-loaded code remain in the cluster, which would prevent the class-loader from unloading code, and repeated loading of classes during development might hit limitations of the permanent generation memory of the JVM. The latter one should not be a problem in real-world scenarios where code is considered to be production stable and only loaded once and rarely unloaded.

These case studies do not address other requirements. The datasets did not reach sizes where resource limitations of single computers made data processing infeasible. Thus the claimed *scalability* features of *DynamoGraph* were not under test. This is why in the following sections two large-scale temporal graphs were used to test the capability of *DynamoGraph* to scale out horizontally as larger graphs are processed.

6.4.1 PageRank over Enron e-Mail Database

As first test-drives to *DynamoGraph* the Enron e-Mail database was used as input dataset. This dataset is just small enough that it would still be possible to compute over a preprocessed version of the data on a single machine with significant amount of memory. The experiments described in the following were computed on the private cloud testbed described in section 6.3.1 with a work-in-progress version of *DynamoGraph* which was still missing much of the API functionality available today and also lacking some optimisations in data-structure access which have a negative performance impact. The results presented in this section were discussed as first result papers in [111] with extended results available as a Journal Paper [105].

The setup of the test-runs presented here is loading the Enron e-Mail database to *DynamoGraph* and then computing *PageRank* over a significant timespan (Jan 2002 to Dec 2003 most of the mailboxes have traffic) of the loaded data. Such that the system is forced to aggregate a large amount of edges prior being able to perform actual computation. Since the same timespan is used repeatedly after the first superstep of *PageRank* edge aggregations are already available in caching data structures such that consecutive supersteps will execute significantly faster. Obviously the assumption is

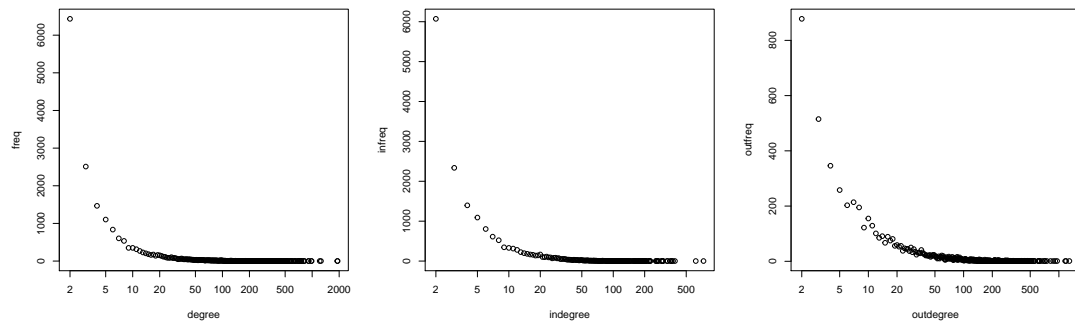


Figure 6.2: Degree distributions for combined, in- and vertex out-degree

that adding more compute resources (processors) to the system will lead to *PageRank* completing faster.

Before studying the results presented in figure 6.3 it is important to notice, that the baseline shown in this paper (P1, 1 slot, 1 worker) is a non-optimised version of the exact same algorithm used in the parallel and distributed runs. The single worker would have a global view of the graph and would not need to perform the message passing overhead. Since there is only one worker thread system resources such as the message router, access to the key value store, and un-necessary partition-table lookups cause bottle-necks.

Especially the message router is implemented as a background thread such that message routing can be performed in parallel to local processing. However, the locking mechanisms in this background thread perform particularly bad in the P1 case. By default the Java API uses non-fair locks, the JVM favours keeping a lock with a certain thread over handing control to another thread and causing a processor context switch. If the P1 case is observed with profilers such as JVisualVM⁷ it becomes clear that message queues fill up to a maximum until a context switch is enforced and a consuming behavior can be started. This behavior needs to be addressed in future versions of the framework for instance the local developer mode of *DynamoGraph* runs as P1 configuration. Further, it is important to state that the configuration 1 slot on 1 worker is also clearly performing worse than a more intelligent sequential implementation of *PageRank*. The speedups given in the following need to be seen in the context of this restrictions and are relative to this P1 baseline.

It comes with no surprise that the execution of *PageRank* for a configuration of 1 slot and 1 worker (see P1 in figure 6.3) is clearly slower (1585 seconds) than for a configuration of 4 slots and 4 workers (P16, 126 seconds) which amounts to a speedup

⁷Oracle Java JVisualVM: <http://docs.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html>

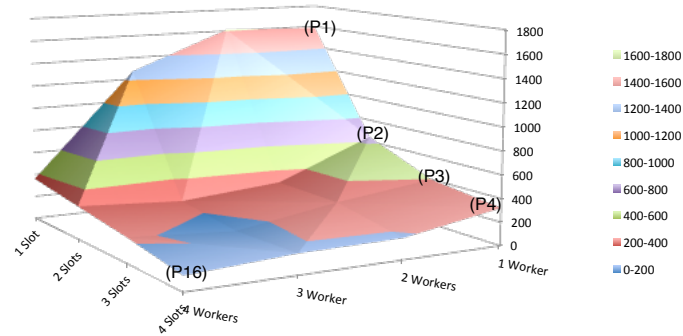


Figure 6.3: Average run times of the PageRank algorithm

of 12.57. This would be a super-scalar speedup, but this is only achieved due to the restrictions stated before.

The experiments were run on a two node infrastructure cloud using a round-robin virtual machine scheduler. This resulted in configurations where virtual machines were bound to use Ethernet network communication and configurations where they could communicate via virtual networking on the host machine. This results in an effect visible in the runs for 1 slot and for 3 slots. The runs with 1 slot and 2 worker nodes is surprisingly slightly slower than the run with 1 slot 1 worker (P1). This is because in this configuration the virtual machines were always hosted on different physical compute nodes. The same effect occurs for the runs with 3 slots where the run with 4 workers is performing slightly better than the run with 3 workers.

From the results it is also clear that running many slots on a single multi-core processor is performing better than running a smaller number of slots distributed over many workers. This becomes obvious when one compares the timings for the run of 1 slot on 1 worker (P1, 1585 seconds), 2 slots on 1 worker (P2, 640 seconds, speedup 2.47), and 4 slots on 1 worker (P4, 321 seconds, speedup 4.93). Slots are executed in the context of the same JVM. Message passing between partitions that are hosted in the same JVM is performed through local `LinkedQueue` implementations which removes the networking overhead.

The results also show that there is in fact performance gain in running algorithms in distributed mode on many compute nodes. For configurations of 2 slots per node the single worker configuration runs in 640 seconds compared to a 3 worker configuration which runs in 231 seconds which amounts to a speedup of 2.77. The situation is very similar in the case of a 4 slot configuration where the single worker run takes 321 seconds and the 4 worker configuration finishes in 126 seconds; a speedup of 2.54.

The results show that for algorithms like *PageRank* that can be formalised in a fully distributed manner good performance gains are possible when actually executed in a

cloud environment. Applied on temporal graphs in a social network analysis scenario a framework like *DynamoGraph* can provide a scalable foundation layer that make real world datasets controllable.

6.4.2 Multiple Algorithms on IU Click Data

Since already at earlier stages of the project, it could be shown that shorter execution times can be achieved by increasing the cluster size, a natural next move is to assess behavior with significantly larger datasets. The hypothesis was that due to the structure of the processing framework *DynamoGraph* clusters can be horizontally scaled to accommodate to dataset sizes. In general multiple large graph datasets for scientific research can be found. Most of the popular graph datasets are organised in the Koblenz Network Collection⁸ and the Stanford Large Network Dataset Collection⁹ [68]. The latter with a clear focus on providing networks of significant size. The datasets found in these two collections are large in the sense that naive algorithms will fail to compute in feasible time. However, they mostly do not impose the restriction of making them unmanageable on traditional computing systems.

A dataset of a dimension that makes processing on traditional single machines infeasible is the Click Dataset from the Center for Complex Networks and Systems Research (CNetS) at Indiana University Bloomington [74]. The dataset is a click stream which is stored as in a proprietary binary format. It can be interpreted as a very large edge list of web-pages referring to other sites. The raw and uncompressed edge-list amounts for approximately 14TB, which is clearly larger than current commodity hardware machines provide in random access memory. However, storage backends and network attached storage systems can easily host data of that size. The dataset and the process to obtain access is described in greater detail in appendix B.3.

As described earlier in general *DynamoGraph* clusters can be rolled out on top of infrastructure clouds. Since the evaluation testbed again has some resource restrictions the Click Dataset was preprocessed using a state of the art Hadoop cluster¹⁰. The goal of the preprocessing was to condense the dataset to a size that allows *DynamoGraph* to hold the complete model in memory on the given private cloud. It was important that an in-memory processing approach can be used to eliminate possible effects from bottle necks in data input / output of the underlying storage system. To achieve this a Map-Reduce [23] job condensed the edge list, which was available at a time resolution of milliseconds, to monthly resolution. Individual edges between vertices were reduced to a single edge and the edge weight was the summed weights of the original edges.

⁸The Koblenz Network Collection: <http://konect.uni-koblenz.de>

⁹Stanford Large Network Dataset Collection: <https://snap.stanford.edu/data/>

¹⁰Apache Hadoop: <http://hadoop.apache.org>

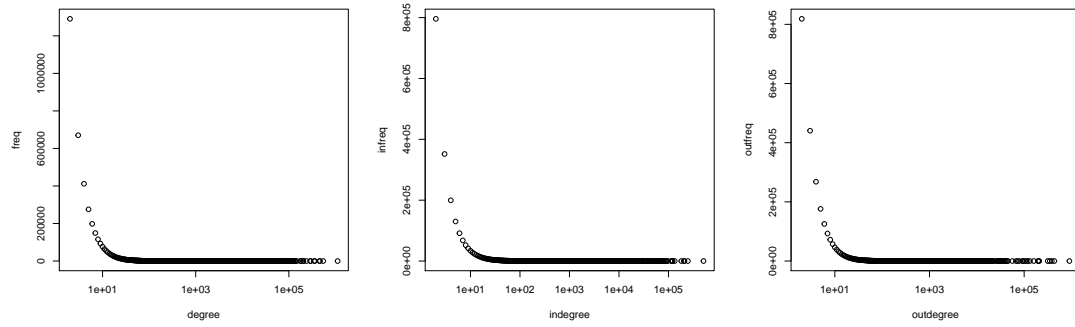


Figure 6.4: Degree distributions for combined, in- and vertex out-degree

Further instead of individual URLs, that denoted vertices in the original graph, also vertices were condensed to DNS domain names. Meaning that e.g. the URLs `http://www.tk.jku.at/teaching` and `http://www.tk.jku.at/research` were condensed to a single vertex named `tk.jku.at`. This resulted in a 93 million edges *DynamoGraph* CSV file that still can fit in memory on the hardware at disposal.

To simulate commodity hardware on the cloud stack a resource configuration typically found in current desktop / laptop machines was used. The virtual machines were equipped with 4 CPUs and 16 GB of memory. Additionally every machine was provided with 40 GB of local storage and 16 GB of storage for swap. Then automated tests were executed and the run-times for dataset import, and a mix of algorithms was recorded. As a first observation we were able to record that for configurations of clusters with less than 7 machines the runs fail altogether. Individual compute nodes run out of memory and crash; the run can not complete.

In the range of available cloud resources many possible configurations were run. The general limiting factors where physical memory and processors of the host machines. Processors are abstracted by the cloud stack such that vast over-provisioning is possible giving the impression only memory limits the size of computing infrastructure. The used virtual machine configuration with 16 GB of memory allowed for up to 18 virtual machines on the 288 GB of RAM provided by the infrastructure cloud. But it allows to simulate 512 virtual CPUs with the hardware at disposal. As documented in the results in this section the limitation of 48 physical CPUs in the system becomes visible as a performance inhibitor. In first trials also other configurations with smaller virtual machines were run. Performance in general decreased. The system required more network communication overhead. Clearly vertex messages buffered in memory also tapped into system memory which lead to memory consumption patterns that involved high usage of swap memory. In extreme cases on very small machines (2GB memory) runtimes of several days were expected. To make testing feasible 8 machines with 16 GB of memory formed the lower bound.

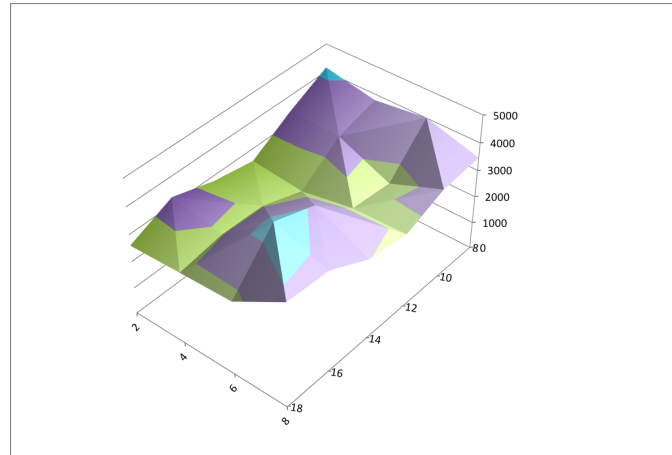


Figure 6.5: Run-times for Click dataset import

In the automated tests cluster configurations built from 8, 10, 12, 14, 16, and 18 virtual machines were used. For each of these configurations the worker machines were configured to run with 2, 4, 6, and 8 threads. Which in the smallest case provides 16 parallel processing threads as opposed to 144 threads. The latter case is clearly a vast overbooking of the 48 physical processor cores installed in the test hardware.

For each run configuration above, three consecutive executions were performed and the average time of the three runs is given in the results below. Each run was given two tasks (1) the Click dataset was imported from an HTTP source on the cluster, and then (2) PageRank was computed over the complete available timespan in a sliding window of 6 month which was moved by one month per iteration. Task (1) is very input/output intensive as up to 144 threads in parallel perform network and disk read operations on a HTTP server and organize the retrieved data in memory structures. (2) in contrast is CPU intensive and requires the system to repeatedly reorganize cached temporal maps.

In the workload of the import task three extreme cases are interesting to observe (see also the 3D plot in figure 6.5). The quickest import was achieved in 39 minutes 20 seconds (2360 seconds) on average with the 12 node 4 slots configuration. This is a natural behaviour since in this configuration all 48 processors installed can be utilized. On larger clusters import time decreases due to limitations in shared network bandwidth. For instance the 16 node 6 slots configuration with 96 processors is 2 times over-provisioning the provided hardware and gives the slowest average import performance of 1 hour 13 minutes 45 seconds. One can see that as soon as more than 48 threads are concerned with the import task performance decreases. This is also the case for scaling to smaller clusters not fully utilizing the available processors such that the configuration of 8 nodes and 2 slots is the slowest, considering the system is not

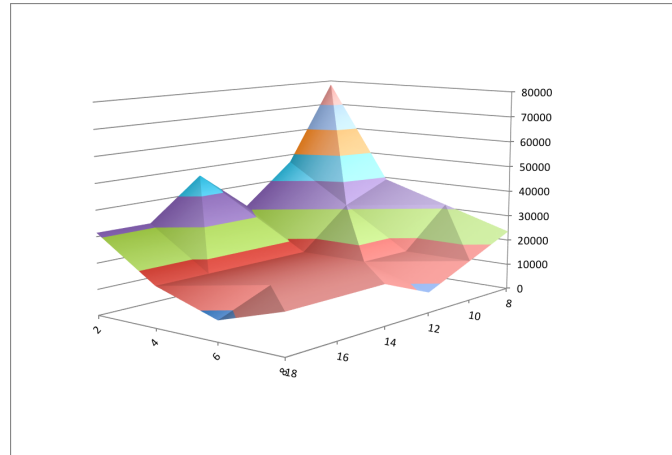


Figure 6.6: Run-times for Click PageRank runs

over-provisioned. In this configuration the import can be done in 1 hour 9 minutes 6 seconds on average.

However, more interesting is whether or not the same scaling effects can be observed that have already been shown with the Enron dataset in section 6.4.1. Runs due to the significantly larger dataset are of course incomparable with the experiments conducted with the Enron dataset. In figure 6.6 a 3D plot of the PageRank runs over the Click dataset are given. From this plot the expected behavior can be observed, adding more compute resources to the problem results in faster execution times.

Again the speedup curve is initially very steep (see 8 virtual machines with 2 slots compared to 8 VMs with 4 slots and 12 VMs with 2 slots). In the extremely small case (8 VMs, 2 slots) an enormous number of vertices is handled on each partition. In separate profiling runs the CPU and memory behavior for these selected configurations was observed. We see that memory pressure during algorithm execution reaches the limits of physical memory assigned to the VM such that the operating system is forced to swap memory blocks to disk. The high memory utilization is triggered by vertex to vertex messages that have to be queued on the receiving partition until superstep execution advances by a step to process the queue. The diagram shows execution times with the curve from 2 to 4 slots being slightly steeper compared to the case moving from 8 to 10 VMs. This is clear since the larger slot configuration adds 16 threads to the configuration where the larger cluster configuration only adds 2.

Also interesting to observe is that in this significantly larger dataset the resource limitations of the underlying hardware infrastructure is reached in certain cases such that the scale-up in larger setups advance in plateaus. A first large plateau is reached in multiple configurations (10 VMs 8 slots, 10 VMs 6 slots, 12 VMs 6 slots, 12 VMs 4 slots, 14 VMs 8-6-4 slots, etc.). An execution time of 4 hours 21 minutes on average marks a

first resource boundary, the installation has only 48 processors which when monitored in the hardware in these configurations are fully utilized during the local processing phases of the superstep. Message queues for vertex to vertex messages do not fully fit into memory but swapping operations can mostly be done in background. Only cache misses in the graph data-structures that require the system to rebuild certain timespan selections again and again are inhibiting factors.

These runs are then only outperformed by the configurations that are using almost all the underlying cloud infrastructure stack's resources. These are the configurations of 18 VMs running 6 slots and 12 VMs running 8 slots. They both complete PageRank on average in 2 hours and 10 minutes. It is clear that in both cases all the 48 physical CPUs available in the system must be fully utilized. Each of the processors is over-provisioned by a factor 2 to 2.25. It seems that the workload is a good mix of processing and IO tasks such that threads that stall for memory and IO can give way for other threads waiting for the processor. Interesting to observe is that these two cases different overall memory configurations. The 12 VMs configuration can utilize 192 GB of RAM and the 18 VMs configuration 288 GB (which is the maximum available amount of memory in the installation). Interestingly the KVM hypervisor used in the OpenStack installation is responsible for some of the observed effect. In the 12 VMs configuration very large partitions still hit the 16 GB resource boundary forcing the VMs operating system to swap. However, the KVM integration (guest and host) in the Linux kernel leads to a situation where the host operating system is capable of completely caching the swapped blocks in RAM such. Cache-miss operations in the guest VM have the impression that memory blocks were retrieved from disk where they are actually read from memory. This behavior is observable in the host machines which constantly reports filled disk caches. In contrast the 18 VMs configuration runs completely without swapping any of the *DynamoGraph* workload and is expected to perform well.

Interestingly technically larger configurations i.e. the maximum of 18 VMs running 8 slots each (144 threads using 288 GB or RAM) perform worse. In this case CPUs are over-provisioned by a factor of 3 and memory is also slightly over limit. Since all real memory is consumed by virtual machines the overheads such as the host-kernel and VM management information, create a slight over-consumption, part of which is compensated for by the host operating system with techniques of data-deduplication. Nevertheless we see that swap space fills in this configuration thus disk IO causing additional stalls in processing.

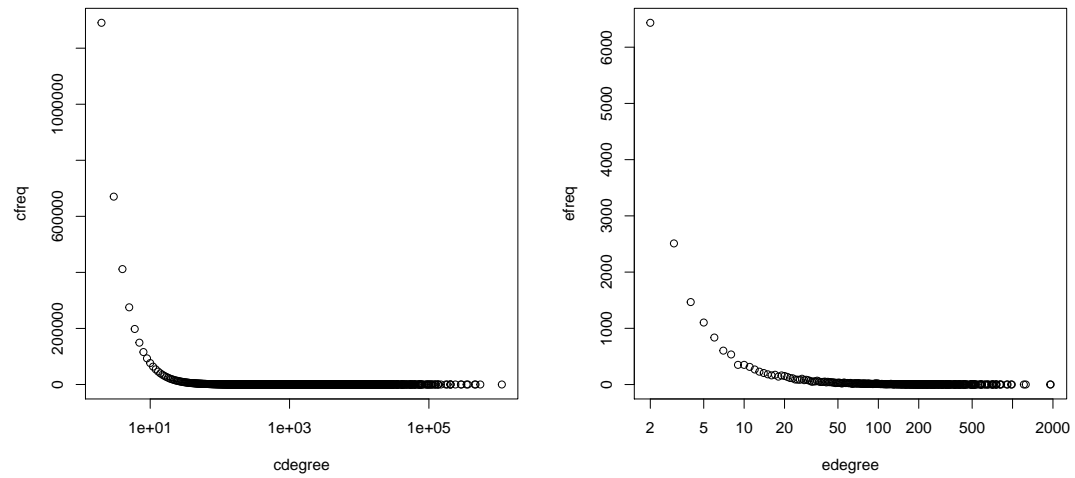


Figure 6.7: Degree distribution of the Click dataset (cdegree) compared with the Enron dataset (edegree)

6.4.3 Social Networks vs. Click Graphs

The two evaluation datasets presented in section 6.4.1 (Enron email database) and in section 6.4.2 represent candidates of network data from two very different application domains. The Enron email database is a case of a communication network which can be categorized as a social network. It is formed through social interaction between mostly human actors. In contrast the Click dataset is a web graph which is a technical network or information network. From the mere metrics it is significant that the studied social network is by magnitudes smaller compared to the web graph. However, this cannot be seen as a general distinguishing criteria in general social networks can grow to very large sizes it is only the case that datasets with dense temporal information are not available to the general scientific audience.

In figure 6.7 the degree distribution charts (x-axis in log-scale) from the last sections are repeated and put in context to each other. It is clear that the degree distribution follows a steeper curve for the web graph compared to the social network. This means that in comparison with the rest of the dataset only a small fraction of the vertices have very high vertex degree and the vast majority of all vertices has insignificant vertex degree for the web-graph. This is in contrast to the social network where the curve shows that there are still a significant number of vertices with degrees between 20 and 50. If the scales are observed it is also obvious that maximum vertex degree sizes are at round 2000 for the social network and in the millions for the web graph.

The observed behavior is natural to the described graphs and the upper bound for vertex degree can be explained in the social network due to effects known as Dunbar's number [26]. Dunbar claims that humans have a natural limit of social bonds they can engage into which in his work is explained by the capabilities of the human neuro-cortex. But the number itself can also be observed in other social networks as in that there seems to be an upper bound on other actors with whom one is able to engage into meaningful relationships [36, 87]. This is what also becomes visible in the Enron dataset, it is very likely that the vertices with very high vertex degree serve as communication hubs and thus do not have very tight relationships with all peers they are interfacing with.

Obviously this upper bound on link capacity does not hold for technical networks as the web graph. Although it is safe to assume that a web-site with a very high in-degree will face a great amount of web traffic and thus will have other resource requirements compared to a web-site with low in-degree, there exist technical solutions to overcome capacity bottlenecks.

But this effect of different vertex-degree distributions in these different classes of network also has implications on the underlying processing architecture. In the vertex-centric computing approach presented in this thesis a natural way of passing on results is to spread local intermediate values to vertex neighbors. This mechanism has been explained in very much detail in chapter 3 and is visible the example algorithms for PageRank (see section 3.7.3) and the label propagation community detection (section 3.7.4). Vertices receiving messages from neighbors usually need to process them in a loop. It is clear from the distributions given, that vertices with significantly larger in-degree will require far longer for processing compared to the average vertex with relatively low in-degree. Since the bulk-synchronous processing model requires to wait until all vertices have completed computation it is a usual scenario that threads and processors have to stall until these high degree vertices have completed.

It is clear that this stalling behavior is more distinctive in the case of the technical networks compared to the social network. In order to tone down this effect counter-measures in the algorithms can be introduced. For instance the PageRank implementation used in the evaluation runs of this thesis requires the outgoing PageRank to overcome a certain configurable threshold before it is propagated to vertex neighbors. Extraordinary small PageRank fractions (i.e. the PageRank portion webpages would get from links in high volume search engines such as Google) will not contribute to make a certain vertex distinguishable from other vertices. While formally this changes the algorithm behavior, it is safe to assume that the introduced error is very small. Almost all vertices in the web graph can be reached from a search engine.

A general solution to this problem can be to introduce better partitioning of data in the system. Future approaches (see section 7.1) will partition edges over the available compute resources as opposed to vertices. Thus evenly distributing compute resources along the cost driving dimension.

6.5 Discussion

During the various phases of evaluation certain interesting aspects can already be formulated. Firstly, the case studies and other student projects which are based in *DynamoGraph* have already shown that the overall engineering approach is feasible. Considering that, for many other distributed computing approaches also significant overhead in learning programming frameworks need to be taken, the Pregel-inspired, vertex centric programming paradigm used in this thesis is feasible. In contrast to more general distributed programming models the vertex centric model used in this case serves a single purpose only. This makes *DynamoGraph* (barely) unusable for other computing problems, but formalising algorithms in the context of a single vertex completely hides erroneous parts of distributed computing (message passing, data distribution, etc.) from programmers.

The software engineers working with *DynamoGraph* have most often been implementing and testing their code in local developer mode with mock data. The most often re-occurring error pattern are jobs without or with badly formulated halt conditions. As already stated earlier this can lead to jobs running infinitely and in the current implementation of *DynamoGraph* will block cluster resources. In some cases malicious halt-conditions did hold on the mock data but broke on real datasets which results in very strenuous debug cycles where worst case the log files of multiple worker nodes need to be examined.

Clearly for production quality software the *DynamoGraph* framework will need further improvements (central log repository, maximum runtime boundaries). However, these can be considered as pure engineering tasks and are beyond the scope of this thesis.

Further evaluation of the prototype clearly shows that, given the still limited compute resources of the available cloud stacks, clear speedups can be achieved by scaling the compute system up vertically and horizontally. Unsurprisingly, vertical scaling has greater impact as opposed to horizontal scaling which has higher cost of inter process communication. In both cases it is clear that the speedup is so significant that a distributed computing approach must always be considered when designing large-scale (temporal) graph applications.

It has also been shown that the approach scales very well to various dataset sizes. Beginning from demo data, typically with below 100 vertices, over to the case studies where around hundreds of vertices make up a graph, up to the large-scale datasets processed during evaluation that range in the dimension of 600.000 email messages for Enron or 53.5 billion HTTP request headers for the Click dataset, all of which can be addressed with the same scalable temporal graph data processing approach.

With the reference implementation, its use in real world case studies and the final evaluation, also the initial hypothesis that a novel software system can tackle real world workloads for large-scale temporal graph processing are fulfilled. It has been shown that:

Temporal Graph Partitioning along the temporal dimension, in what is called temporal resolution in this thesis, and structurally using traditional graph partitioning methods serves the purpose well. The experimental datasets used in the studies clearly follow the expected behavior that the growth in a graphs diameter is faster than the growth in time dimension.

Distributed Temporal Graph Storage within this work was realized using temporal maps. These can hold self contained copies of temporal-graph vertices. They seem to be an optimal foundation for the vertex-centric processing approach used.

Distributed Temporal Graph Processing was designed around industry proven distributed computing concepts. It has been shown that the vertex-oriented Pregel processing paradigm can as well be applied to temporal-graph structures.

Chapter 7

Conclusions and Future Work

In the following future directions of the presented approach are discussed in detail and a conclusion section provides a wrap up of the contributions made in this thesis.

7.1 Future Work

The distributed computing methods in context of temporal graph processing are a lively topic of research in various disciplines right now. In providing a distributed computing approach to solve large scale problems in this area an important contribution was made, which allows for shorter question, answer, reformulation cycles in real world scenarios. However, the presented approach comes at the cost to run extensive computing infrastructure.

On the other hand many areas of the addressed area of research remain untouched by this thesis or new questions arise from the current state of research. In the following some prospective topics for future work are highlighted and briefly discussed.

7.1.1 Visualisation and Tooling

One key aspect that requires major improvement is the software stack itself. While substantial effort was made to create a research prototype mainly resource restrictions hindered to push this prototype to production quality. Functionality obviously needed in enterprise settings such as authentication and the configuration of access restrictions, strategies for long-term data backup, and more straightforward configuration are still missing.

Some of the code in *DynamoGraph* implements functionality already available elsewhere as open source library. For instance the current trend for Big Data processing and high-performance computing has given rise for complete software frameworks that support tasks such as cluster management and intra cluster message passing. One such prospective framework is JGroups¹. It supports like organising multiple compute nodes and their threads in clusters. Further it provides a transparent mechanism for intra-cluster message passing which is implemented with plain network sockets in *DynamoGraph*. JGroups proposes that their message passing implementation can transparently integrate with enterprise message passing systems such as the Java Messaging System (JMS). It is yet to be tested if the overhead a system like JGroups introduces has noticeable performance impact on *DynamoGraph*. Vast amounts of network traffic can be handled in background such that minor throughput degradation will very likely not have high impact on the system.

Further, the tooling for visualisation is still in alpha stadium. In *DynamoGraph* the SigmaJS² library is used for web-based graph drawing and Highcharts³ for arbitrary other charts. First test with various other modern JavaScript UI libraries such as D3.js⁴ seem to be good candidates to base production ready visualisation on. D3.js is widely used in the data-analysis industry and would allow to draw graphs and other charts on the same technological foundations.

7.1.2 Performance Improvements

In this thesis the general scalability of the approach is shown. With this it is clear that with certain restrictions adding more computing resources to a compute cluster will allow us to compute over larger input datasets or allows us to compute metrics for a given input dataset in shorter time on a larger cluster. Fast execution is a key point.

The method presented here follows state of the art distributed programming models. In recent related work it has been shown that there is still much room for improvement performance wise. Mainly two strands of optimization are proposed: (1) instead of the intuitive approach of applying edge-cuts when partitioning a graph vertex-cuts can be used, and (2) for large parts of graph computations either structural information or factual information of the vertices is used, that is either vertices and edges or their attributes are in the focus of computation [37]. In (1) the results presented in this thesis undermine the idea that vertex-cuts could improve performance in the sense that a more homogenous dispersion of data over compute resources can be reached. Thus

¹JGroups: <http://jgroups.org/>

²SigmaJS: <http://sigmajs.org/>

³Highcharts: <http://highcharts.com/>

⁴D3.js - Data Driven Documents: <https://d3js.org>

effects of stalling processors can be downscaled (see also the discussion in chapter 6, section 6.4.2).

In the discussed temporal graph models where a vertex is seen as a temporal document or temporal map with the edges in the role of a vertex attribute, vertex-cuts are much harder to achieve. Much of the attributes of a vertex will reside on multiple partitions in the system and thus data manipulation operations will naturally become more difficult, error-prone, and slower. A vertex is not a singular object but a complex document.

To overcome this and to also address the already mentioned improvement stand (2) from above a decoupling of data attributes and graph structural data can be done. This would mean that for the graph itself pure time-stamped edge lists could serve as a very simple data model. Whereas all the other information about vertices and edges can be stored in (temporal) documents on state of the arte key-value storage engines. This means that the attribute model is isolated from the network model and for each of the two facets individual optimization strategies can be used.

For the edges it is clear, vertex-cuts and thus evenly distributing edges over compute resources can provide a performance improvement. For attributes well known document retrieval strategies with multilayered caches can be implemented. In the *DynamoGraph* project both changes would require substantial changes in the data and processing architecture such that these improvements need to be moved to later iterations of the project.

7.1.3 Dynamic Notion of Time

A final improvement that can be made to the presented approach is addressing the notion of time in use. Currently data is structured in the time dimension in what is called the temporal *resolution*. This means that data in vertices is grouped in time-frames of fixed size in a certain time unit. For instance if data is represented in hourly resolution then time-frames of the length of one hour are the smallest unit for data processing.

DynamoGraph in its implementation already allows time-framed queries such that data from multiple such fixed width blocks can be combined. A developer can run algorithms in arbitrarily long time-windows of a datasets resolution.

However, when observing real world datasets it becomes clear that information is represented in varying granularity over time. For instance networks derived from business e-mail databases will show higher event density during work hours but less density

during night and on weekends. Thus it would be preferable to allow for dynamicity in specifying time frames for data storage and processing.

In *DynamoGraph* the concept of a timeframe is interwoven with many concepts of the underlying data structure, the temporal map. In general the storage frameworks requires all notions of time to be aligned along time resolutions. This allows for the system to simply compute fitting timeframes from points in time over comparing a point in time with multiple timeframe candidates, which is a search task. Implementing lookup mechanisms in the storage backend that divert from the static resolution based model would require substantial adaption and thus retesting of basic project infrastructure.

7.1.4 Incremental Computation

Related work on dynamic graph algorithms [29] also shows that the temporal graph opens new interesting strands for research. In *DynamoGraph* (see also section 6.4.2 in chapter 6) a typical use case is to compute a certain metric over a sliding window. This allows to reason over the development of certain metrics over time. The downside of this approach is that large computational overhead is taken to recompute values and intermediate values which were already available when processing data in preceeding time windows. This is especially the case for networks where the degree of change can be predicted as particularly low (i.e. road networks, global telephone network).

It would be interesting to research as to whether it is possible for certain of these algorithms to derive the result of a timeframe T_n from the already computed results of a preceeding timeframe T_{n-1} . The assumption would be that for many algorithms this is in fact possible and could substantially reduce the computational overhead.

Further, for algorithms where the above stated hypothesis will not hold it could still be interesting if results from previous timeframes can be used as the seed for consecutive algorithm runs. For instance algorithms such as PageRank initialize all vertices with a predefined constant initial rank. It is yet to be confirmed if such metrics can be computed in faster time in sufficient accuracy by using past PageRank values as the initial values for recomputation.

7.2 Concluding Remarks

The presented work documents the results of research in the area of large-scale temporal graphs. It was motivated that the formal construct of a graph which retains time

information as it changes has practical relevance and thus applied research in this topic is valuable. Further, it was highlighted that the most interesting problems that can be modelled as temporal graphs usually are very large-scale problems ultimately making compute models and their concrete implementations a necessity to advance research in this area.

In this work then preliminaries in the field of graphs and temporal graphs in particular were given to illustrate which formal models and important algorithms for them exist. Prospective application areas were discussed and from these possible requirements for a novel temporal graph processing system were derived. The hypothesis was made that by finding suitable distributed data structures, and providing simple query and processing mechanisms for application developers will allow to build a temporal graph processing system that can serve as the foundation for future research but also applications on real world networked problems.

From a literature review in relevant areas it was concluded that none of the existing approaches are suitable for the exact requirements derived in this thesis. But it was also clear that certain aspects of this work can be rooted in existing paradigms such as the Pregel graph processing approach [73] and with this a vertex-centric data distribution method.

An extensive research prototype, called *DynamoGraph*, was implemented. It fulfills many of the posed requirements and demonstrates that the proposed solution, of using distributed temporal maps with temporal selecting mechanisms in interplay with adapted and extended versions of Pregel, is a technically viable approach. On top of *DynamoGraph* several case-studies and a performance evaluation was conducted to prove the point that the framework is usable in real-world scenarios. It was shown that the datastructures in use, the algorithm formalisation method and job execution are constructed in a way that allows third-party application programmers to base their systems on *DynamoGraph* and it was further shown that the distributed computing approach is scalable and flexible enough that vertical and horizontal scale-up scenarios can be used to improve application performance or to address larger problems. These results are underpinned by experimental performance evaluations.

With this it is clear that temporal large-scale graphs are a class of data that occur in various applications and thus has a high practical relevance. With frameworks like *DynamoGraph* they become manageable and thus they can be used in real world application scenarios. Still the research field provides large potential such as more intelligent partitioning schemes (vertex- over edge-splits) and the use of dynamic graph algorithms.

Appendix A

Tooling

The real world implementation of this thesis available as open source software. The code, detailed instructions, and code samples can be found at <http://www.dynamograph.net/>. The following chapter gives a brief overview of the tools in use, lists their versions, and purpose they were used for.

A.1 Java, JavaScript, Web

As highlighted in chapter 4 the prototype implementation was mainly implemented with the Java programming language. There are basically two components worth mentioning in this respect. The first component is the computing cluster which is implemented as a standalone distributed Java application that can be deployed to possibly many worker nodes in a computer cluster. The second component is the web based user interface which is implemented as a Java Enterprise web application that runs in a Servlet container. The the following paragraphs and sections describe the tools and versions used in these components.

For software project management in general and automated dependency management the infamous Apache Maven 3.1.1 toolset is used. All components of the prototype are available as a Maven configured artifacts and can be automatically built and tested on state-of-the-art software integration services such as Jenkins <http://jenkins-ci.org>. Although continuous integration is currently performed from the developers machines since the project team was rather small and most of the time consisted only of the thesis author.

Although Java 8 was already available at the time of writing this the prototype was still compiled and tested with Java 1.7.0_45-b18 since first tests with Java 8 VMs have shown that some minor changes in object serialization could cause problems with the

distributed compute cluster and the new features available in Java 8 were not used by the project anyway.

The following list shows all Maven projects in alphabetical order and their purpose is briefly explained.

analytics: The analytics project is a helper project that contains Java and R code for general statistical analysis of datasets and is not available on production installations of the system.

census: This contains lists of firstname and lastname, and their frequency of occurrence in the 1990 USA census [117]. The project contains a library that can be used to assign unique firstname, lastname pairs to vertices which is used to assign random names to make anonymized datasets. This process has proven to make anonymized datasets better readable.

commandline: This project contains command-line utilities that allow some control over a running temporal graph processing cluster. Mainly it can be used to properly shut down a cluster and to perform other maintenance tasks.

common: This project contains general data-structures to describe a temporal graph such as vertices and edges and also contains more general data-structures such as the implementation of a temporal map which is the basic foundation of most data components in the system.

configuration: As the name suggests this project is just a helper that allows to read configuration files from disk and provides the loaded configuration as static variables to the many other components of the system.

datasources: This project provides libraries that enable access to diverse data-sources such as static data-sets that hold temporal edge lists, relational databases, e-mail database parsers, and dynamic data-sources such as directly streaming data from Twitter.

distributed: This resembles the core of the distributed compute cluster and holds the code for all components such as the master node, the worker node, election process, graph partition table, and all the communication and multi-threading code. Implementation details of this project are described in greater detail in A.1.1.

dynamo-graph: This is an almost empty Maven parent project. It just holds a Maven project object model that can be used to build, test, and deploy all system components in one coherent build process.

frontend: Here the web based user interface is implemented. The project can be built to a standard Java EE web application contained in a WAR file and deployable to any standard compliant Servlet container as explained in more detail in A.1.3.

opencv: This project contains code that allows to reconstruct past social networks from image data. Such that user provided picture libraries can be imported and a temporal social network is extracted from them.

playground: The playground project is a potpourri of different code snippets that were used for testing. To the interested reader this project might be interesting because it shows how code can be run on top of the distributed computing framework. Especially interesting might be the parts that allow developers to run the system in so called local developer mode to debug distributed algorithms.

staticdata: This is a project that contains static data-sets used during testing and evaluating this thesis. The data-sets are mostly compiled from temporal edge lists. Each data-set available to the system also contains a configuration file that can be parsed by the frontend application and contains information like a data-set description, the color coding used and the edge types contained in the data-set.

uml: This project contains ObjectAid 1.1.6 UML class diagrams that are automatically synced with the code in the other projects. The UML diagrams are used for documentation purposes in this thesis and on the project website.

utils: Finally the utils project contains general purpose utilities that did not fit with any other project but are used throughout many components of the project. Such utilities are an extended version of an URL resolver, parser for files encoded as comma separated values, and a tool that is capable of merging colors available as RGB color codes.

A.1.1 Distributed Java Applications

As already explained in great detail the distributed graph computing framework is implemented as a standalone Java application. However, projects of such high complexity are not possible without high quality frameworks that support certain aspects of the

systems requirement. In the case of *DynamoGraph* the following open source projects deserve to be mentioned.

Apache ZooKeeper is an open source configuration database that supports resilient, distributed configuration. It is a system which can be used to implement distributed synchronisation mechanisms. The project is particular useful since many of the best practices in distributed computing are very well documented in recipes¹. This takes much of the complexity involved in creating a distributed system away from a software developer and moves responsibilities such as synchronisation and locking into a library which is very well tested and industry proven.

Other complex tasks are also widely based on ZooKeeper recipes in other systems such as Apache Spark and Apache Hadoop. For instance the leader election implemented for a failsafe *DynamoGraph* cluster are strictly implemented after the provided guidelines.

Gson, Jackson, FST serialisation libraries are used extensively to free software developers from implementing network protocols. In general *DynamoGraph* uses a binary protocols between its compute nodes. These binary protocols are implemented with Java object serialisation mechanisms. The mechanisms provided with the Java API are known to be rather slow such that the FST Library² was implemented. The generated serialized objects are binary compatible with Java API serialisation. However, certain circumstances (e.g. single messages exceeding 100MB) might require that users configure *DynamoGraph* to use standard serialisation.

For debugging purposes it was sometimes necessary to being able for humans to interpret the data on network channels. This can be achieved by enabling JSON serialisation instead of binary protocols. Depending on the JSON library available on the classpath either the Jackson³ or the Gson⁴ library can be used.

A.1.2 Client API Methods

This section gives a brief overview of the available functions in the *DynamoGraph* client API as provided by the master node. The methods are roughly categorized by tasks.

Namespace Management: With these commands namespaces can be managed.

¹ZooKeeper Recipes: <https://zookeeper.apache.org/doc/trunk/recipes.html>

²Fast Serialisation (FST): <https://ruedigermoeller.github.io/fast-serialization/>

³FasterXML Jackson JSON library: <https://github.com/FasterXML/jackson>

⁴Google Gson library: <https://github.com/google/gson>

```
List<Vertex> listModels()  
boolean containsModel(String namespace)
```

The methods `listModels` and `containsModel` can be used to reason about the currently instantiated namespaces in a cluster.

```
void createModel(String namespace, Resolution res)  
void deleteModel(String namespace)
```

These methods can be used to manipulate the list of models configured at a cluster. Needless to state that any data not saved somewhere else is lost upon namespace deletion. For the model creation a resolution needs to be specified. All points in time and timespans in the model will automatically resolved to this resolution.

```
Timeframe getMaximumTimeframe(String namespace)  
ModelDescriptor getModelDescriptor(String namespace)
```

The functions `getMaximumTimeframe` and `getModelDescriptor` can be used to retrieve detailed information about *DynamoGraph* namespaces. The maximum timeframe function can be used to determine the start and end time of all the data stored in a namespace. A model description contains information about a models resolution, its name and a description.

Graph Manipulation Instructions: Are all commands that allow the software developer to create, update and delete individual components of the graph. Technically speaking manipulating graph elements in a namespace.

```
void addNode(String namespace, Vertex vertex)
```

With this method the application developer can prepare a `Vertex` object in her program and load it into a namespace. The method throws an error if the vertex (according to vertex id) already exists in the namespace.

```
void deleteVertex(String namespace, long vertexId)  
void deleteVertex(String namespace, Vertex vertex)
```

This method can be used to delete a vertex. It is possible to either pass the vertex id or a `Vertex` object as a parameter.

```
void addEdge(String namespace, Edge edge, long time)
```

An `Edge` object is in general a timestamped edge with two fields for the source and target id for the connected vertices. By calling this method the edge is automatically inserted in the in-edge and out-edge collections of the affected vertices.

```
Vertex getVertex(String namespace, long vertexId)
Vertex getVertex(String namespace, long vertexId, Timeframe tf)
```

With this method developers can retrieve a `Vertex` object from the cluster, given its vertex id. Optionally a timeframe can be specified thus requiring that the vertex exists (has attributes) in the denoted timeframe.

```
void updateVertex(String namespace, Vertex updatedVertex)
```

With this method a vertex and all its attributes can be updated. It is good practice to retrieve a copy of the vertex from the cluster prior updating. The `updateVertex` method assumes that the vertex exists and overwrites it without further sanity checks.

```
long findMaxVertexId(String namespace)
```

Calling this will return a `long` denoting the largest vertex id found in the given namespace. This can be used to determine a valid, unused vertex id for new vertices that are about to be added to a namespace.

```
Vertex findVertex(String namespace, String vertexName)
Vertex findVertex(String namespace, String vertexName, Timeframe tf)
```

These two functions can be used to find a vertex by their name attribute. Technically no direct lookup table for names exists such that a graph query (`queryVertices`) is built on the master and its result is returned. See details in the next block.

Superstep Algorithm Execution: This contains a set of methods that can be used to schedule algorithm execution, allows to query current status of algorithm execution, and helps in managing custom user code on the cluster.

```
long executeAlgorithm(String namespace, String clazz)
long executeAlgorithm(String namespace, String clazz,
    SuperStepContext context)
long executeAlgorithm(String namespace, String clazz,
    SuperStepContext context, Timeframe tf)
```

The `executeAlgorithm` method instructs the compute cluster to execute the Superstep denoted by the `clazz` parameter over the specified namespace. Optionally a `SuperStepContext` can be provided which might contain configuration values for the algorithm. Further an optional timeframe can restrict as in which timeframe the algorithm is to be executed. The return value of these calls is a long denoting the superstep profile id. This id is used to query details about algorithms with the methods described in the following.

```
SuperStepExecutionProfile getExecutionProfile(long profileId)
```

A `SuperStepExecutionProfile` describes an instance of a Superstep in great detail. It is used by the master node to track execution status and consequently can be used by application developers to infer current execution status. The most important field in this object is the status field of type `ExecutionProfileState`, it denotes the current overall state of the Superstep (New, Executing, Waiting, Completed, and Failed). But the profile contains much more information such as the number of current active vertices, the number of already executed phases and steps, timing information, the number of pending vertex-to-vertex messages, and the last copy of the global algorithm memory. Some of the attributes can be queried with the shorthand methods in the following.

```
ExecutionProfileState waitForAlgorithmState(long profileId,  
                                             ExecutionProfileState... states)  
ExecutionProfileState waitForCompletion(long profileId)
```

With `waitForAlgorithmState` an application developer can make a client application stall until the superstep on the master reaches a certain state. This way a client application can submit a job, then perhaps makes some further instructions as to track meta-information about the job. And then wait for the superstep to switch into Completed or Failed state. The method `waitForCompletion` is a shorthand for this behavior it stalls until the algorithm completes or fails.

```
SuperStepContext queryAlgorithmContext(long profileId)
```

This is a shorthand to query only the global memory from a profile.

```
ExecutionProfileState queryAlgorithmState(long profileId)
```

This to query only the profile state.

```
Exception queryExceptions(long profileId)
```


And this can be used to retrieve exceptions that caused an algorithm run to fail. The method will return null on all profiles that are not in Failed state.

```
List<Vertex> queryVertices(String namespace, GraphQuery query)
```

Graph queries are special forms of supersteps that can return a list of vertices as their result. They are used to retrieve more complex queries back from a *DynamoGraph* cluster. For instance a vertex and its neighbours or vertices with certain attributes.

```
void loadCode(String clazz, byte[] binaryJar)
void unloadCode(String clazz)
```

The `loadCode` and `unloadCode` instructions can be used to allow software developers to upload their custom superstep implementations to a cluster. The parameter `clazz` is used to identify the main class of the superstep and is also used to identify the uploaded code package. The code itself must be encoded as a standard Java JAR file. The content of this JAR file must be supplied in the `byte[]` which is the second parameter to the function.

Bulk Data Import: This group of instructions allows a software developer to bulk import data from datasources which support remote import.

```
long startRemoteImport(ConfiguredModelProvider provider,
                        String namespace)
RemoteImportProgressStatus queryImportProgress(long importJobId)
```

The method `startRemoteImport` launches an import stop using the configured model provider which basically describes the datasource to be used. The import target is the already existing namespaces denoted with the second parameter. The method returns a job id which can be used with the method `queryImportProgress` to query the cluster on the progress of the import. It returns a complex object which holds many details about the import such as the number of already imported vertices opposed to the expected number of elements still in the queue.

Cluster Status and Monitoring:

```
ClusterInfo getClusterInfo()
```

Cluster info can be queried from a *DynamoGraph* master to reason about the current system status. The cluster info contains information about the number of active work-

ers, number of active partitions, and whether or not the cluster is currently active and ready to process commands.

`Sample benchMarking()`

With the benchmarking function more detailed information about the cluster can be retrieved. However, depending on configuration it might be the case that no benchmarking metrics are recorded. A benchmarking sample holds the latest system metrics for the master and worker nodes. This can be basic hardware specific information such as the current CPU load and memory pressure but in the case of a master sample also contains information about the currently executing supersteps.

A.1.3 Web-based User Interface

As explained in 5 users working with a system like the presented will most likely prefer to use web-based user interfaces. The high volume of data do be processed is just one argument for web-based access which practically allows the user to have the data completely stored in the cloud and just interface the processing from a web-browser. In the case of this project state-of-the-art web technology was used to build the user interface. The application is built as Java web application compliant with version 3.0 of the Servlet, version 2.2 of the JSP (Java Server Pages), and 2.2 of the EL (Expression Language) specifications. The visual rendering of the web pages is based on the JSF (Java Server Faces) 2.1.7 implementation provided by PrimeFaces 3.3.1 which guarantees that the user interface also complies with current standards for cross platform web-design.

For graph visualization the frontend relies on the JavaScript based graph drawing library SigmaJS 0.1 which was extended by a few simple modules which allow the user to move vertices in the graph via drag and drop, enable context information for each vertex such that computed scores for each vertex can be displayed, and the users are able to directly interact with vertices to drill deeper in a graph when necessary.

In general the web-based user interface can be built via Maven by running a build in the *frontend* project. The result is a web application archive (WAR) that can be deployed on any arbitrary Servlet 3.0 container. During the implementation and testing of this thesis mainly Apache Tomcat 7.0.29 and earlier was used and thus Apache Tomcat is highly recommend for production environments also.

Other open source frameworks used to implement the web frontend of *DynamoGraph* are jQuery and Highcharts.js mainly to reder chart data.

A.2 Cloud Stacks

DynamoGraph is designed to run on cloud stacks. This is for two reasons, firstly, cloud computing is the current paradigm for harnessing infrastructure resources. To allow *DynamoGraph* to integrate well with state of the art software stacks it was a natural choice to also base it on the same stack. Secondly, it was shown that the processing architecture scales well which means that in real world application scenarios large software clusters need to be run. It is a cumbersome task to setup and maintain a large number of computers with the same software stack such that the mechanisms provided by infrastructure cloud stacks is very helpful.

In early stages of the project an OpenNebula⁵ cloud stack was used as the foundational layer. This stack was later dismissed due to other workloads using the same stack required changes in the network topology. Evaluation of several stacks have shown that the OpenStack⁶ provides enough flexibility to host several projects on the same infrastructure while providing extensive network isolation. Further OpenStack provides wide compatibility to public cloud stacks such as the Amazon AWS system and provides integration for state of the art infrastructure provisioning systems such as Chef⁷ and Puppet⁸.

For convenience reasons, mainly because the OpenStack cloud stack itself was already provisioned and configured using Puppet also the *DynamoGraph* nodes are provisioned with a Puppet manifest. All the experiments conducted were run on top of the OpenStack release 2014.2.2 codename Juno. The Puppet infrastructure runs off of version 3.8.1.

⁵OpenNebula: <http://opennebula.org>

⁶OpenStack: <https://www.openstack.org>

⁷Chef: <https://www.chef.io/chef/>

⁸Puppet: <https://puppet.com>

Appendix B

Used Datasets

Whilst developing, testing, and evaluating this thesis many different datasets from different sources were used. In this section all the datasets used are listed, their licensing, source, and usage is explained in greater detail. The datasets are generally divided in artificial and natural datasets that were collected from real world observations. Algorithms usable to generate artificial datasets are briefly described in 6.2.2 and were seldom used.

For the real world datasets two important factors were considered (1) data should be generated by humans or some natural process, such that the data is a model of a real world network, and (2) the dataset needs to be of significant size to demonstrate that horizontal scalability can be used to scale to large data sizes.

In the following three datasets and their source are discussed in greater detail to allow the interested reader to obtain the same data for related research.

B.1 MIT Reality Commons

The MIT Human Dynamics Lab has a long history in researching human signals. It is a very active group around Alex (Sandy) Pentland, the author of *Honest Signals* [90]. In their efforts of analyzing many aspects of human behavior they also created several data sets.

These datasets are of different age and depending on the technological advancements in mobile computing the collection methods varied over time. At the beginning so called sociometric badges were used to capture user interaction. These badges are wearable electronic devices that are capable of measuring face-to-face interaction between humans. They record each others infrared signals in order to detect face-to-face

interaction and audio features in order to detect activity in these interactions. Later feature phones and smart phones were used in the sensing process, completely replacing the sociometric badges. The effort of collecting data on mobile phones eventually resulted in the funf framework [2], which is a highly configurable platform for measuring all sorts of sensor data of a mobile phone.

All datasets that are available in the Reality Commons contain a variety of different sensor readings and also related information like performance indicators or music genre preferences. In this work we mainly focus on information that we can use to reconstruct a dynamic social network of the sensed communities. The individual datasets are described in the following mainly focusing on how they can be used to construct a social network.

The first dataset labeled Reality Mining [27] was created in 2004 and contains Bluetooth proximity, phone call logs and text message logs collected from Symbian based mobile phones. This dataset contains sensor readings from 100 subjects that participated in the study for at least a year. Obviously the data can be used to construct a social network that shows the three different dimensions of interaction: face-to-face, phone calls and text messages.

Later the Badge [88] dataset was collected. It was created using the sociometric badges described earlier. These badges were developed since the mobile phones were proven to be not feasible in a workspace environment where users did not carry their phones on them at all times. So in this study sociometric badges were worn in the workspace in order to detect face-to-face interaction and certain other data about the users. The big advantage of this dataset is the fact that also individual work performance data was collected. Performance was measured by a ticketing system that was used to handle inbound customer requests. This way it is possible to predict which social patterns lead to high work performance.

In 2010 a large study on Friends and Family [3] was conducted. Approximately 64 families and 130 individual participants were taking part. This dataset was also collected on mobile phones and contains data about face-to-face interaction derived from bluetooth proximity, voice calls and text messages which allow to construct a dynamic social network. Further this dataset contains ground truth about couples, closeness between individuals and certain other facts about private life that can be used to check results.

And finally the last dataset available in the Reality Commons collection is the Social Evolution Dataset [72] collected in the years 2008 and 2009. It contains data from everyday life of a whole undergraduate dormitory. With the collection of this dataset

the diffusion of information, opinions and illnesses was measured. This dataset was also recorded with mobile phones but contains large samples of opinions, symptoms, and preferences that were collected by repeated online surveys during the study.

All files of the MIT Reality Commons are available as edge list files. These are comma separated value (CSV) files with a common structure. The first column usually denotes the unique source id of a person, the second column the target id, followed by a column for the edge weight (if any) and a date or timestamp. The DynamoGraph platform supports direct import of CSV files through the dataset import mechanisms described in section 5.5. The datasets described above are relatively small the largest (Friends and Family) amounts for 300MB in total.

B.2 Enron e-Mail database

In October 2001 in the US a big scandal around tax fraud and stock market manipulation involving the Enron Corporation was uncovered which eventually led to bankruptcy of the enterprise [10]. During the investigations and the aftermath of this case large amounts of Enron's e-mail communication was acquired by the Federal Energy Regulatory Commission. This data set further on called the Enron e-Mail corpus was then sold to Andrew McCallum at the University of Massachusetts Amherst who made his copy generally available to research [20]. This is why this e-mail corpus is widely used in studies for social network analysis and natural language processing.

This corpus is unique in that respect that it is the only available corpus which contains a very large number of real e-mails for a very long time period. The oldest messages in the corpus date back to 2001 and the newest one were sent in 2004. The original data set that was released by the University of Massachusetts contained approximately 600.000 messages. The original data source however was revised in 2010 by Electronic Discovery Reference Model LCC (EDRM)¹ and manual data cleansing was applied such that in the EDRM dataset [57, 20] 1.7 million messages can be found. In the EDRM corpus the data cleansing process was far more precise such that only around 10.000 items had to be deleted. Mainly messages which contained sensitive information such as credit card numbers, social security numbers, national identity numbers and passport numbers, individuals dates of birth, and information of highly personal nature such as medical and legal matters. The cleansed EDRM dataset is of 1.43 size and is mainly interesting because it contains real human-to-human interaction.

¹Electronic Discovery Reference Model LCC: <http://edrm.net/>

In the EDRM download all emails are available as maildir mailboxes in plain MBox format. Which means all e-mail contents with the complete header information and the e-mail body are stored in individual files. As for DynamoGraph only the social network inherent in the dataset were of research interest a converter application was created. The converter is able to ingest the e-mail database. During this process e-mail addresses found in the database are cleansed and filtered. Then vertex id's are assigned to each e-mail and for each occurrence of an e-mail sender sending a message to a recipient an edge is created. The generated data is stored to CSV files which can be directly imported to the DynamoGraph platform through the available importers (see section 5.5).

B.3 Click Dataset

Available through the Center for Complex Networks and Systems Research (CNetS) at Indiana University Bloomington the Click Dataset is currently the largest web traffic network available for research. The data was collected as part of research conducted by Weiss et al. [74]. The datasets contains 53.5 billion HTTP requests of HTTP traffic from and to Indiana University. The HTTP header information is stripped down to the bare minimum information necessary to reconstruct a web graph. It contains the requested URL, the referring URL, and two boolean flags. One indicates whether or not traffic was user or bot generated, and the other denotes the origin of the request (inside or outside of Indiana University).

Data was collected on a packet filter on an edge-router of the university and contains data from September 2006 to May 2010 with 275 days missing. The data is broken down into hourly files and files for any individual day are compressed using tape archive (TAR) and GZip compression. The data files themselves are in a custom line based format which mixes binary and text formats. The first line contains the timestamp and flags in binary format followed by the referrer URL. Line two and three contain the host-name and the path of the request URL.

The dataset can be obtained from CNETS under licensing and disclosure conditions found on their website². Technically the data is made available in encrypted and compressed format on a single *3TB* hard drive. Uncompressed the data amounts for more than *14TB* of data.

In order to import the data to our DynamoGraph test-cluster available at the Institute of Telecooperation significant preprocessing of the data was required. The data was

²CNETS Click Dataset: <http://cnets.indiana.edu/groups/nan/webtraffic/click-dataset/>

decrypted and decompressed on the fly, individual request lines were filtered and the domain names of request and referrer were extracted. This data was then stored as edge lists in CSV files which again can be easily imported to the *DynamoGraph* platform.

Bibliography

- [1] The JSON Data Interchange Format. Technical report, ECMA International, October 2013.
- [2] N. Aharony and W. Gardner. Funf Developer Site. <http://www.funf.org>, 2012.
- [3] N. Aharony, W. Pan, C. Ip, I. Khayal, and A. Pentland. Social fMRI: Investigating and Shaping Social Mechanisms in the Real World. pages 643–659, December 2011.
- [4] E. C. Akrida, L. Gašieniec, G. B. Mertzios, and P. Spirakis. Ephemeral Networks with Random Availability of Links: The Case of Fast Networks. *Journal of Parallel and Distributed Computing*, 87:109–120, 2016.
- [5] R. Albert and A. L. Barabási. Statistical Mechanics of Complex Networks. *Reviews of Modern Physics*, 74(47), January 2002.
- [6] L. A. N. Amaral, A. Scala, M. Barthelemy, and H. E. Stanley. Classes of Small-World Networks. *Proceedings of the National Academy of Sciences*, 97(21):11149–11152, September 2000.
- [7] A. Amelio and C. Pizzuti. Analyzing Voting Behavior in Italian Parliament: Group Cohesion and Evolution. In *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*, pages 140–146, Aug 2012.
- [8] W. E. Baker and R. R. Faulkner. The Social Organization of Conspiracy: Illegal Networks in the Heavy Electrical Equipment Industry. *American Sociological Review*, pages 837–860, 1993.
- [9] M. Baur, U. Brandes, M. Gaertler, and D. Wagner. Drawing the AS Graph in 2.5 Dimensions. In *Graph Drawing*, pages 43–48. Springer Berlin Heidelberg, Berlin, Heidelberg, September 2004.

-
- [10] BBC News Online. Enron Scandal At-a-Glance. <http://news.bbc.co.uk/2/hi/business/1780075.stm>, 08 2002.
 - [11] T. Y. Berger-Wolf and J. Saia. A Framework for Analysis of Dynamic Social Networks. In *Proceedings of the 12th ACM SIGKDD*, 2006.
 - [12] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *The Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, New York, New York, USA, June 1996. ACM.
 - [13] P. Boldi, M. Santini, and S. Vigna. A Large Time-Aware Web Graph. *ACM SIGIR Forum*, 42(2):78–83, November 2008.
 - [14] U. Brandes. A Faster Algorithm for Betweenness Centrality*. *Journal of Mathematical Sociology*, 25(2):163–177, August 2010.
 - [15] C. Cattuto, M. Quagiotto, A. Panisson, and A. Averbuch. Time-varying Social Networks in a Graph Database: a Neo4j Use Case. In *Graph Data-management Experiences & Systems*, pages 11–6, New York, New York, USA, June 2013. ACM.
 - [16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
 - [17] A. C.-L. Chen, M. Elzohbi, A. Elhajj, K. F. Xylogiannopoulos, P. Karampelas, M. Ridley, and R. Alhajj. Finding Hidden Nodes by Checking for Missing Nodes Against Past Social Networks. In *2012 15th International Conference on Network-Based Information Systems (NBIS)*, pages 239–246. IEEE, 2012.
 - [18] C.-Y. Chen, A. Ho, H.-Y. Huang, H.-F. Juan, and H.-C. Huang. Dissecting the Human Protein-Protein Interaction Network via Phylogenetic Decomposition. *Scientific Reports*, 4:7153–10, November 2014.
 - [19] A. Clauset, M. Newman, and C. Moore. Finding Community Structure in Very Large Networks. *Physical Review E*, 70(6):066111, December 2004.
 - [20] W. W. Cohen. Enron Email Dataset. <http://www.cs.cmu.edu/~enron/>, 08 2009.

-
- [21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms: Graph Algorithms. In *24.3 Dijkstras Algorithm*. MIT Press, 2009.
- [22] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *ACM SIGPLAN Notices*, pages 1–12. ACM, July 1993.
- [23] J. Dean and S. Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107–113, January 2008.
- [24] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [25] G. Di Fatta, F. Blasa, S. Cafiero, and G. Fortino. Fault Tolerant Decentralised K-Means Clustering for Asynchronous Large-Scale Networks. *Journal of Parallel and Distributed Computing*, 73(3):317–329, March 2013.
- [26] R. I. M. Dunbar. Neocortex Size as a Constraint on Group Size in Primates. *Journal of Human Evolution*, 22(6):469–493, June 1992.
- [27] N. Eagle and A. Pentland. Reality Mining: Sensing Complex Social Systems. *Personal and Ubiquitous Computing*, 2006.
- [28] R. Ecker. Creation of Internet Relay Chat Nicknames and Their Usage in English Chatroom Discourse. *Linguistik Online*, 50(6):108c–108c, 2011.
- [29] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic Graph Algorithms. In *Algorithms and Theory of Computation Handbook*. Springer, 1998.
- [30] P. Erdős and A. Rényi. On Random Graphs I. In *Publicationes Mathematicae*, pages 290–297. Institute of Mathematics, University of Debrecen, Hungary, 1959.
- [31] D. A. Ferrucci. IBM’s Watson/DeepQA. In *ISCA ’11: Proceedings of the 38th annual international symposium on Computer architecture*. IBM, ACM, June 2011.
- [32] M. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, 21(9):948–960, 1972.

- [33] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. In *25th Annual Symposium on Foundations of Computer Science*, pages 338–346. IEEE, 1984.
- [34] M. Girvan. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, June 2002.
- [35] P. Gloor, R. Laubacher, Y. Zhao, and S. Dynes. Temporal Visualization and Analysis of Social Networks. In *NAACSOS Conference*, 2004.
- [36] B. Gonçalves, N. Perra, and A. Vespignani. Modeling Users’ Activity on Twitter Networks: Validation of Dunbar’s Number. *PLOS ONE*, 6(8):e22656, August 2011.
- [37] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX Conference on Operating System Design and Implementation*, 2012.
- [38] M. C. González, C. A Hidalgo, and A.-L. Barabási. Understanding Individual Human Mobility Patterns. *Nature*, 453(7196):779–782, June 2008.
- [39] B. Göschlberger. *A Platform for Social Microlearning*, pages 513–516. Springer International Publishing, Cham, 2016.
- [40] O. Green, R. McColl, and D. A. Bader. A Fast Algorithm for Streaming Betweenness Centrality. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Conference on Social Computing (SocialCom)*, pages 11–20. IEEE, 2012.
- [41] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI. Portable Parallel Programming with the Message-Passing Interface*. Massachusetts Institute of Technology, 1999.
- [42] W. Hant, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos. In *EuroSys*, New York, USA, 2014. ACM Press.
- [43] F. Harary and G. Gupta. Dynamic Graph Models. *Mathl. Comput. Modelling*, 25(7):79–87, 1997.
- [44] B. W. Herr, W. Ke, E. Hardy, and K. Borner. Movies and Actors: Mapping the Internet Movie Database. In *Information Visualisation*, pages 465–469. IEEE Computer Society, 2007.

- [45] L.-Y. Ho, J.-J. Wu, and P. Liu. Distributed Graph Database for Large-Scale Social Computing. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 455–462. IEEE, 2012.
- [46] S. H. Hong, N. S. Nikolov, and A. Tarassov. A 2.5D Hierarchical Drawing of Directed Graphs. *Journal of Graph Algorithms and Applications*, 11(2):371–396, 2007.
- [47] J. Hopcroft, O. Khan, B. Kulis, and B. Selman. Natural Communities in Large Linked Networks. In *KDD '03: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 541–546, New York, USA, August 2003.
- [48] C. L. Hsu, S. J. Park, and H. W. Park. Political Discourse Among Key Twitter Users: The Case of Sejong City in South Korea. *Journal of Contemporary Eastern Asia*, 12(1):65–79, 2013.
- [49] A. Jacobs. The Pathologies of Big Data. *Communications of the ACM*, 52(8):36, August 2009.
- [50] M. Jacomy, S. Heymann, T. Venturini, and M. Bastian. ForceAtlas2, A Graph Layout Algorithm for Handy Network Visualization. Technical report, Gephi Consortium, August 2011.
- [51] M. Janssen, Y. Charalabidis, and A. Zuiderwijk. Benefits, Adoption Barriers and Myths of Open Data and Open Government. *Information Systems Management*, 29(4):258–268, October 2012.
- [52] U. Kang, D. H. Chau, and C. Faloutsos. Mining Large Graphs: Algorithms, Inference, and Discoveries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 243–254, 2011.
- [53] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *2009 Ninth IEEE International Conference on Data Mining (ICDM)*, pages 229–238. IEEE, 2009.
- [54] G. Karypis and V. Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–13. IEEE Computer Society, 1998.

- [55] D. Kempe, J. Kleinberg, and A. Kumar. Connectivity and Inference Problems for Temporal Networks. *International Journal of Computer and System Sciences*, 64(4):820–842, 2002.
- [56] H. Kim and R. Anderson. Temporal Node Centrality in Complex Networks. *Physical Review E*, 85(2):026107, February 2012.
- [57] B. Klimt and Y. Yang. Introducing the Enron Corpus. Technical report, Language Technology Institute, Carnegie Mellon University, 2009.
- [58] V. Kostakos. Temporal Graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.
- [59] G. Kotsis. *Interconnection Topologies and Routing for Parallel Processing Systems*. PhD thesis, University of Vienna, 1991.
- [60] F. Kuhn and R. Oshman. Dynamic Networks: Models and Algorithms. *ACM SIGACT News*, 42(1):82–96, March 2011.
- [61] Jérôme Kunegis. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on the World Wide Web*, 2013.
- [62] M. J. Kushin and K. Kitchener. Getting Political on Social Network Sites: Exploring Online Political Discourse on Facebook. *First Monday*, 2009.
- [63] A. Kwan and L. Bic. A Structuring Technique for Compute-Aggregate-Broadcast Algorithms on Distributed Memory Computers. In *Sixth International Parallel Processing Symposium*, pages 10–17. IEEE, 1992.
- [64] A. Lancichinetti, S. Fortunato, and J. Kertész. Detecting the Overlapping and Hierarchical Community Structure in Complex Networks. *New Journal of Physics*, 2009.
- [65] D. Laney. 3D Data Management: Controlling Data Volume, Velocity and Variety. Technical report, Gartner, 2001.
- [66] J K Laurila, D Gatica-Perez, and I Aad. The mobile data challenge: Big data for mobile computing research. *Nokia Mobile Data Challenge*, 2012.
- [67] Harold J Leavitt. Some effects of certain communication patterns on group performance. *Journal of Abnormal and Social Psychology*, 46(1):38–50, 1951.

- [68] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [69] Y.-R. Lin, H. Sundaram, and A. Kelliher. Multi-Relational Characterization of Dynamic Social Network Communities. In Borko Furht, editor, *Handbook of Social Network Technologies and Applications*, pages 1–30. Springer US, Boston, MA, 2010.
- [70] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(01):5–20, 2007.
- [71] C. Lynch. Big Data: How Do Your Data Grow? *Nature*, 455(7209):28–29, 2008.
- [72] A. Madan, M. Cebrian, S. Moturu, K. Farrahi, and A. Pentland. Sensing the ‘Health State’ of a Community. *IEEE Pervasive Computing*, 11(4):36–45, 2011.
- [73] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD International Conference on Management of Data*, 2010.
- [74] M. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani. Ranking web sites with real user traffic. In *Proc. First ACM International Conference on Web Search and Data Mining (WSDM)*, pages 65–75, 2008.
- [75] M. R. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani. Ranking Web Sites with Real User Traffic. In *Proceedings of the First ACM International Conference on Web Search and Data Mining*, pages 65–76, New York, New York, USA, February 2008. ACM.
- [76] P. Mell and T. Grance. The NIST Definition of Cloud Computing (Draft). *NIST Special Publication*, 800:145, 2011.
- [77] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM TOS*, 11(3):14–34, July 2015.
- [78] V. Z. Moffitt and J. Stoyanovich. Towards a Distributed Infrastructure for Evolving Graph Analytics. In *Proceedings of the 25th International Conference Companion on World Wide Web*, Montréal, Canada, 2016.

- [79] C. Moler. Matrix Computation on Distributed Memory Multiprocessors. *Hypercube Multiprocessors*, 86:181–195, 1986.
- [80] J. Mondal and A. Deshpande. Managing Large Dynamic Graphs Efficiently. In *ACM SIGMOD International Conference on Management of Data*, 2012.
- [81] J. L. Moreno. *Who Shall Survive*. Beacon House Inc., 1934.
- [82] P. Mucha, T. Richardson, K. Macom, M. Porter, and J.-P. Onnela. Community Structure in Time-Dependent, Multiscale, and Multiplex Networks. *Science*, 328(5980):876–878, May 2010.
- [83] J.-B. Musso. Network Management and Impact Analysis with Neo4j. <https://linkurio.us/network-management-and-impact-analysis-with-neo4j/>, April 2014.
- [84] NetworkX. NetworkX: Graph Generators. <https://networkx.github.io/documentation/latest/reference/generators.html>.
- [85] M. Newman and M. Girvan. Finding and Evaluating Community Structure in Networks. *Physical Review E*, 69(2):026113, February 2004.
- [86] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora. Graph Metrics for Temporal Networks. In *Temporal Networks*, pages 1–27. Springer, January 2014.
- [87] Don’t Believe Facebook; You Only Have 150 Friends. <http://www.npr.org/2011/06/04/136723316/dont-believe-facebook-you-only-have-150-friends>.
- [88] D. O. Olguín, B. N. Waber, Taemie K., A. Mohan, K. Ara, and A. Pentland. Sensible Organizations: Technology and Methodology for Automatically Measuring Organizational Behavior. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(1):43–55, 2009.
- [89] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, 1999.
- [90] A. Pentland. *Honest Signals: How They Shape Our World*. MIT Press, August 2008.

- [91] M. A. Porter, P. J. Mucha, M. E. J. Newman, and C. M. Warmbrand. A Network Analysis of Committees in the U.S. House of Representatives. *Proceedings of the National Academy of Sciences*, 102(20):7057–7062, May 2005.
- [92] D. Pucher. Layered Visualisation for Large-Scale Temporal Graphs in Dynamo-Graph. Master’s thesis, Johannes Kepler University Linz, August 2016.
- [93] L. Quick, P. Wilkinson, and D. Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, ASONAM ’12, pages 457–463, Washington, DC, USA, 2012. IEEE Computer Society.
- [94] U. N. Raghavan, A. Réka, and S. Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3):036106, September 2007.
- [95] Z. Richardson. Golden Orb: Open Source Pregel Implementation. <https://github.com/jzachr/goldenorb>.
- [96] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM: Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM Request Permissions, July 2013.
- [97] S. E. Schaeffer. Graph Clustering. *Computer Science Review*, 1:27–64, 2007.
- [98] R. Schindler. Grundprinzipien der psychodynamik in der gruppe. In *Psyche*, 1957.
- [99] T. Schiphorst. Affectionate Computing: Can We Fall in Love with a Machine? *IEEE Multimedia*, 13(1):20–23, January 2006.
- [100] A. Schrijver. On the History of the Transportation and Maximum Flow Problems. *Mathematical Programming*, 91(3):437–445, 2002.
- [101] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.
- [102] M. Steinbauer. Sensor Based, Automated Detection of Behavioural Stereotypes in Informally Formed Workgroups. Master’s thesis, Johannes Kepler University

Linz, February 2012.

- [103] M. Steinbauer and G. Anderst-Kotsis. Using DynamoGraph: Application Scenarios for Large-scale Temporal Graph Processing. In *17th International Conference on Information Integration and Web-based Applications & Services*, 2015.
- [104] M. Steinbauer and G. Anderst-Kotsis. DynamoGraph: A Distributed System for Large-scale, Temporal Graph Processing, its Implementation and First Observations. In *6th Temporal Web Analytics Workshop, Companion to WWW2016*, 2016.
- [105] M. Steinbauer and G. Anderst-Kotsis. DynamoGraph: Extending the Pregel Paradigm for Large-scale Temporal Graph Processing. *International Journal on Grid and Utility Computing*, 7(2):141–151, 2016.
- [106] M. Steinbauer, M. Hiesmair, and G. Anderst-Kotsis. Making Computers Understand Coalition and Opposition in Parliamentary Democracy. In *Electronic Government*, pages 265–276. Springer International Publishing, Cham, September 2016.
- [107] M. Steinbauer, I. Khalil, and G. Kotsis. Building an On-Demand Virtual Computing Market in Non-Commercial Communities. In *ACM SAC 2013*, pages 1–6, 2013.
- [108] M. Steinbauer, I. Khalil, and G. Kotsis. Reality Mining at the Convergence of Cloud Computing and Mobile Computing. *ERCIM News*, pages 1–3, March 2013.
- [109] M. Steinbauer and G. Kotsis. Building an Information System for Reality Mining Based on Communication Traces. In *15th International Conference on Network-Based Information Systems*, pages 1–5, Melbourne, June 2012.
- [110] M. Steinbauer and G. Kotsis. Platform for General-Purpose Distributed Data-Mining on Large Dynamic Graphs. In *Workshops on Enabling Technologies Infrastructure for Collaborating Enterprises*, Hammamet, Tunisia, 2013.
- [111] M. Steinbauer and G. Kotsis. Towards Cloud-based Distributed Scaleable Processing over Large-scale Temporal Graphs. In *Workshops on Enabling Technologies Infrastructure for Collaborating Enterprises*, 2014.
- [112] H. Sundaram, Y.-R. Lin, M. De Choudhury, and A. Kelliher. Understanding Community Dynamics in Online Social Networks: A Multidisciplinary Review.

- Signal Processing Magazine, IEEE*, 29(2):33–40, March 2012.
- [113] M. I. M. Tabash. Monitoring and Benchmarking Toolchain for the DynamoGraph Framework. Master’s thesis, Johannes Kepler University Linz, February 2014.
- [114] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Temporal Distance Metrics for Social Network Analysis. In *Workshop on Online Social Networks*, pages 1–6, May 2009.
- [115] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Characterising Temporal Distance and Reachability in Mobile and Online Social Networks. In *ACM SIGCOMM Computer Communication Review*, pages 118–124. ACM, January 2010.
- [116] J. K. Tang. *Temporal Network Metrics and Their Application to Real World Networks*. PhD thesis, Robinson College University of Cambridge, December 2011.
- [117] Genealogy Data: Frequently Occuring Surnames from Census 1990. U.S. Census Bureau.
- [118] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.
- [119] D J Watts. Networks, Dynamics, and the Small-World Phenomenon. *American Journal of Sociology*, 105(2):493–527, 1999.
- [120] D. J. Watts. *Small Worlds: The Dynamics of Networks Between Order and Randomness*. Princeton University Press, 1999.
- [121] D. J. Watts and S. H. Strogatz. Collective Dynamics of Small-World Networks. *Nature*, 393(6684):440–442, June 1998.
- [122] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES ’13*, pages 2:1–2:6, New York, NY, USA, 2013. ACM.
- [123] J. Yang and J. Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

-
- [124] W. W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 1977.
 - [125] M. H. Zack. Researching Organizational Systems Using Social Network Analysis. In *33rd Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 2000.
 - [126] M. Zignani, S. Gaito, G. Paolo Rossi, X. Zhao, H. Zheng, and B. Y. Zhao. Link and Triadic Closure Delay: Temporal Metrics for Social Network Dynamics. volume 14, pages 564–573, 2014.



Matthias Steinbauer

Personal Data

Name Matthias Steinbauer
Date of Birth 3rd of January 1983, Linz, Austria
Citizenship Austria
e-Mail matthias@steinbauer.org

Affiliation

University Johannes Kepler University Linz
Faculty Faculty of Engineering and Natural Sciences
Department Institute of Telecooperation
Adress Altenberger Strasse 69, 4040 Linz, Austria
Phone +43 664 7500 3498
www <https://steinbauer.org>

Education

Sept 1997 HTL Leonding, Department for Informatics and Software Engineering
Jun 2002
Oct 2003 Bachelors Program in Computer Science, Johannes Kepler University Linz
Nov 2007
Nov 2007 Masters Program in Computer Science, Johannes Kepler University Linz
Jan 2012
Feb 2012 Doctoral Program in Technical Sciences, Johannes Kepler University Linz
Nov 2016

Work Experience

Aug 1998 Marktgemeinde St. Georgen an der Gusen, Internship as Clerical Assistant
Jul-Dec 1999 Axioma Information Systems GmbH, Internship as Java Software Developer
Aug 2001 Programmierfabrik Hagenberg GmbH, Parttime Software Developer
April 2002
Oct 2002 Compulsory Community Service with the Austrian Red Cross, serving as a Paramedic
Sept 2003
Feb 2003 HNC Consulting GmbH, Software Developer and Consultant for the ARIS Business
Feb 2004 Platform

Mar 2004 IDS Scheer Austria GmbH (Software AG), Senior Consultant for the ARIS Business
 Dec 2008 Platform
 since Jul 2004 Independent Contractor, web-services, web-hosting, Linux based IT infrastructures,
 software development, process consulting
 Mar 2007 Johannes Kepler University Linz, Tutor as support for several lectures at the Institute
 Jun 2007 for Pervasive Computing
 Feb 2008 Solutiongroup Software GmbH, Senior Software Developer and Project-Lead for the
 Jan 2011 immformer.com Start-Up in Real Estate Marketing
 Jan 2009 (Voluntary) Co-Leader of the St. Georgen / Gusen branch of the Austrian Red Cross
 Dec 2013
 Mar 2011 Racon Software GmbH, Senior Software Developer for ELBA Internet, an online
 Feb 2012 banking application for the web and mobile-web
 Mar 2012 Johannes Kepler University Linz, Research and Teaching Assistant at the Institute of
 Sept 2016 Telecooperation
 Mar 2012 Johannes Kepler University Linz, Coordinator for the project *You can make IT*,
 Sept 2016 responsible for PR activities of the Department of Computer Science, Johannes Kepler
 University Linz
 since Mar 2014 University of Applied Sciences Upper Austria, Lecturer on Cross-Platform Development
 of Mobile Applications
 since onlinegroup.at creative online systems, Head of Software Development
 Oct 2016

Teaching Experience

Mar 2004 Regular Trainings in Business Process Design and Management
 Dec 2008
 since First Aid Courses for the Austrian Red Cross
 Nov 2007
 since First Aid for Outdoor Enthusiasts, biyearly course
 Oct 2012
 Mar 2012 Tutorials on Multimedia Systems at the Institute of Telecooperation, Johannes Kepler
 Jul 2012 University Linz
 Oct 2012 Introduction to Cloud Computing at the Institute of Telecooperation, Johannes Kepler
 Jan 2013 University Linz
 Mar 2013 Tutorials on Multimedia Systems at the Institute of Telecooperation, Johannes Kepler
 Jul 2013 University Linz
 Oct 2013 Introduction to Cloud Computing at the Institute of Telecooperation, Johannes Kepler
 Jan 2014 University Linz
 Mar 2014 Tutorials on Multimedia Systems at the Institute of Telecooperation, Johannes Kepler
 Jun 2014 University Linz
 Mar 2014 Cross-Platform Development of Mobile Applications, University of Applied Sciences
 Jun 2014 Upper Austria, Campus Hagenberg
 Mar 2015 Tutorials on Multimedia Systems at the Institute of Telecooperation, Johannes Kepler
 Jun 2015 University Linz

- Mar 2015 Mobile Media Design, Institute of Telecooperation, Johannes Kepler University Linz
Jun 2015
- Mar 2015 Cross-Platform Development of Mobile Applications, University of Applied Sciences
Jun 2015 Upperaustria, Campus Hagenberg
- Oct 2015 Tutorials on Software Development (Intro Class) at the Institute of Pervasive Computing,
Jan 2016 Johannes Kepler University Linz
- Mar 2016 Mobile Media Design, Institute of Telecooperation, Johannes Kepler University Linz
Jun 2016
- Mar 2016 Cross-Platform Development of Mobile Applications, University of Applied Sciences
Jun 2016 Upperaustria, Campus Hagenberg

Co-Supervised Theses

- 2014 Melanie Tomaschko: *Interaktive Mathematik auf mobilen Geräten (Interactive Mathematics on Mobile Devices)* Bachelor Thesis co-supervised with Prof. Dr. Gabriele Anderst-Kotsis
- 2014 Orif Orifov: *Context Based, Mobile Learning in the Cloud* Master Thesis co-supervised with Prof. Dr. Gabriele Anderst-Kotsis
- 2014 Mo'min I.M. Tabash *Monitoring and Benchmarking Toolchain for the DynamoGraph Framework* Master Thesis co-supervised with Prof. Maria Carla Calzarossa and Prof. Dr. Gabriele Anderst-Kotsis
- 2015 Mathias Raab: *Drupal als Basis für moderne Konferenz-Webseiten (Drupal as core layer for modern conference websites)* Bachelor Thesis co-supervised with Prof. Dr. Gabriele Anderst-Kotsis
- 2015 Patrick Wall: *MOOCs in Higher Education* Bachelor Thesis co-supervised with Prof. Dr. Gabriele Anderst-Kotsis
- 2015 Mario Meirhuber: *Towards Cloud Computing Interoperability and Transportability for Services on the Platform as a Service Layer / Tier* Master Thesis co-supervised with Prof. Dr. Gabriele Anderst-Kotsis
- 2015 Dominik Prilmüller: *ConnectUs: Using Mobile Applications to Connect Mobile Users on Social Networks* Bachelor Thesis co-supervised with Prof. Dr. Gabriele Anderst-Kotsis
- 2016 Markus Hiesmair: *Understanding the Community Structure of Coalition and Opposition in Parliamentary Democracy on the Example of the Austrian Parliament* Bachelor Thesis co-supervised with Prof. Dr. Gabriele Anderst-Kotsis
- 2016 Simon Angerbauer, Roman Socovka; *Cross-Platform Mobile Game Development (with Unity)* Final Project Polytechnic School Perg, Austria

Publications

- Jan 2012 Matthias Steinbauer, Sensor Based, Automated Detection of Behavioural Stereotypes in Informally Formed Workgroups, Master Thesis, 2012
- Oct 2012 Matthias Steinbauer and Gabriele Anderst-Kotsis, Building an Information System for Reality Mining Based on Communication Traces, In proceedings of *The 12th International Conference on Network-Based Information Systems*, 2012, Melbourne, Australia

- Apr 2013 Matthias Steinbauer, Ismail Khalil, and Gabriele Anderst-Kotsis, Reality Mining at the Convergence of Cloud Computing and Mobile Computing, In *ERCIM News Issue 93*, 2013
- Mar 2013 Matthias Steinbauer, Ismail Khalil, and Gabriele Anderst-Kotsis, Building an On-Demand Virtual Computing Market in Non-Commercial Communities, In proceedings of *The 28th Symposium on Applied Computing*, 2013, Coimbra, Portugal
- Jun 2013 Matthias Steinbauer, and Gabriele Anderst-Kotsis, Platform for General-Purpose Distributed Data-Mining on Large Dynamic Graphs, In proceedings of *22nd Workshops on Enabling Technologies Infrastructure for Collaborative Enterprises*, 2013, Hammamet, Tunisia
- Sept 2013 Matthias Steinbauer, and Gabriele Anderst-Kotsis, Take a Penny, Leave a Penny Scaling out to Off-premise Unused Cloud Resources, In proceedings of *Virtualisierung: gestern, heute und morgen* co-hosted at *INFORMATIK 2013* the 43rd annual meeting of the GI, 2013, Koblenz, Germany
- Sept 2013 Orifjon Orifov, Matthias Steinbauer, Ismail Khalil and Gabriele Anderst-Kotsis, Learning-as-a-Game in the Cloud, In proceedings of *MicroLearning 7.0, 2013, Krems, Austria*
- Dec 2013 Sylva Girtelschmid, Matthias Steinbauer, Vikash Kumar, Anna Fensel, Gabriele Anderst-Kotsis, Big Data in Large Scale Intelligent Smart City Installations, In proceedings of *15th International Conference on Information Integration and Web-based Applications and Services, 2013, Vienna, Austria*
- Apr 2014 Matthias Steinbauer, Ismail Khalil, Gabriele Anderst-Kotsis, Challenges in the Management of Federated Heterogeneous Scientific Clouds, In the *Journal of Integrated Design and Process Science*, 2013
- May 2014 Sylva Girtelschmid, Matthias Steinbauer, Vikash Kumar, Anna Fensel, Gabriele Anderst-Kotsis, On the Application of Big Data in Future Large Scale Intelligent Smart City Installations, *International Journal of Pervasive Computing and Communication*, Emerald (The journal's most downloaded paper 2015)
- May 2014 Matthias Steinbauer, Leitartikel zum Rohstoff Daten, *OCG Journal*, Vol. 39 (1), Österreichische Computer Gesellschaft
- Jun 2014 Matthias Steinbauer, Gabriele Anderst-Kotsis, Towards Cloud-based Distributed Scaleable Processing over Large-scale Temporal Graphs, In proceedings of *23rd Workshops on Enabling Technologies Infrastructure for Collaborative Enterprises*, 2014, Parma, Italy
- June 2015 Yerzhan Baimbetov, Ismail Khalil, Matthias Steinbauer, Gabriele Anderst-Kotsis, Using Big Data for Emotionally Intelligent Mobile Services through Multi-Modal Emotion Recognition, *13th International Conference On Smart Homes and Health Telematics* Geneva, Switzerland
- Dec 2015 Matthias Steinbauer, Gabriele Anderst-Kotsis, Using DynamoGraph: Application Scenarios for Large-scale Temporal Graph Processing, In proceedings of *17th International Conference on Information Integration and Web-based Applications and Services*, 2015, Brussels, Belgium
- 2015 to appear Matthias Steinbauer, Gabriele Anderst-Kotsis, DynamoGraph: Extending the Pregel Paradigm for Large-scale Temporal Graph Processing, *International Journal of Grid and Utility Computing*, Inderscience

- Apr 2016 Matthias Steinbauer, Gabriele Anderst-Kotsis, DynamoGraph: A Distributed System for Large-Scale Temporal Graph Processing, its Implementation and First Observations, *6th Temporal Web Analytics Workshop, Companion to the World Wide Web Conference 2016, Montréal, Canada*
- Sept 2016 Matthias Steinbauer, Markus Hiesmair, Gabriele Anderst-Kotsis: Making Computers Understand Coalition and Opposition in Parliamentary Democracy, *15th IFIP Electronic Government Conference (eGov), 2016, Guimarães, Portugal*
- Nov 2016 Matthias Steinbauer, DynamoGraph: Large-scale Temporal Graph Processing and its Application Scenarios, PhD Thesis, 2016

Awards

- 2013 Best Paper Award for the paper titled: *Big Data in Large Scale Intelligent Smart City Installations*
- 2014 Official recognition by the Major of Vienna to successful international event organisers, accepted as the role of an *Organising Chair* of the *15th International Conference on Information Integration and Web-based Applications & Services* and the *11th International Conference on Advances in Mobile Computing & Multimedia*
- 2015 Award for Excellence, Highly Commended Paper, International Journal of Pervasive Computing and Communications for the paper titled: *On the Application of Big Data in Future Large Scale Intelligent Smart City Installations*
- 2016 Meritorious Paper Award, 15th IFIP Electronic Government Conference (eGov), 2016, Guimarães, Portugal, for the paper titled: *Making Computers Understand Coalition and Opposition in Parliamentary Democracy*

Conference Organizations

15th International Conference on Information Integration and Web-based Applications and Services, 2-4 December 2013, Vienna Austria,
Role: Organising Chair.

11th International Conference on Advances in Mobile Computing & Multimedia, 2-4 December 2013, Vienna Austria,
Role: Organising Chair.

16th International Conference on Information Integration and Web-based Applications and Services, 4-6 December 2014, Hanoi, Vietnam,
Role: Proceedings Chair.

12th International Conference on Advances in Mobile Computing & Multimedia, 8-10 December 2014, Kaohsiung, Taiwan,
Role: Steering committee co-chair.

17th International Conference on Information Integration and Web-based Applications and Services, 11-13 December 2015, Brussels, Belgium,
Role: Proceedings Chair, Organising co-chair.

13th International Conference on Advances in Mobile Computing & Multimedia, 11-13 December 2015, Brussels, Belgium,
Role: Proceedings Chair, Organising co-chair.

Editorial Work

- Dec 2012 David Taniar, Eric Pardede, Matthias Steinbauer, Ismail Khalil *Proceedings of the 14th International Conference on Information Integration and Web-based Applications and Services*
- Dec 2012 David Taniar, Eric Pardede, Matthias Steinbauer, Ismail Khalil *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia*
- Jul 2013 Book Review for the *MIT Press*, Cambridge, Massachusetts, USA
- Dec 2013 Edgar Weippl, Maria Indrawan-Santiago, Matthias Steinbauer, Gabriele Anderst-Kotsis, Ismail Khalil, *Proceedings of the 15th International Conference on Information Integration and Web-based Applications and Services*
- Dec 2013 René Mayrhofer, Liming Luke Chen, Matthias Steinbauer, Gabriele Anderst-Kotsis, Ismail Khalil, *Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia*
- 2014 Liming Luke Chen, René Mayrhofer, Matthias Steinbauer, Guest Editorial for the MoMM 2013 Special Issue, *International Journal of Pervasive Computing and Communication*, Vol. 10, Issue 1, 2014
- Feb 2014 Reviewer for the *Workshops on Enabling Technologies for Collaborating Enterprises*, Parma, Italy, 2014
- Jun 2014 Reviewer for the *5th International Conference on Computer and Communication Technology*, Allahabad, India, 2014
- Jun 2015 Program Committee member of the *Workshops on Enabling Technologies for Collaborating Enterprises* track for *Convergence of Distributed Clouds, Grids and their Management*, Larnaca, Cyprus, 2015
- Nov 2015 Program Committee member of the *10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* track for *Data Storage in Distributed Computation and Cloud Systems*, Krakow, Poland, 2015
- 2015 to appear Editorial for the *International Journal of Pervasive Computing and Communications*, Vol. 11, Issue 2
- Jun 2016 Program Committee member of the *Workshops on Enabling Technologies for Collaborating Enterprises* track for *Convergence of Distributed Clouds, Grids and their Management*, Paris, France, 2016
- Dec 2015 Maria Indrawan-Santiago, Matthias Steinbauer, Ismail Khalil, Gabriele Anderst-Kotsis *Proceedings of the 17th International Conference on Information Integration and Web-based Applications and Services*
- Dec 2015 Liming (Luke) Chen, Matthias Steinbauer, Ismail Khalil, Gabriele Anderst-Kotsis *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia*

Invited Talks and Seminars

- Jan 2014 Continued Education Seminar for Teachers in Higher Education, HTBLA Vöcklabruck, Austria, *New Aspects in e-Learning: eBooks, Microlearning, Gamification, and Collaboration*
- Apr 2014 Edudays 2014, Krems, Austria, *Unterwegs spielend lernen durch Mikrolernen in der Cloud*

- Jun 2014 IDC Datahub 2014, Vienna, Austria, *Parallel Processing of Temporal Graphs in Near-Realtime*
- Nov 2014 OCG Digital 2014, Vienna, Austria, *Applications Scenarios for Large Scale Stream Processing*
- Jan 2015 Continued Education Seminar for Teachers in Higher Education, HTBLA Vöcklabruck, Austria, *New aspects of e-Learning: Cloud Computing and Copyrights in Austrian Schools*
- Mai 2015 IDC Datahub 2015, Vienna, Austria, *Applications Scenarios for Large Scale Stream Processing*
- Nov 2015 Seminars for the IT Department of the Federal State Government of Upper Austria, *RESTful Webservices in Practice*
- Jan 2016 Continued Education Seminar for Teachers in Higher Education, HTBLA Vöcklabruck, Austria, *New Aspects in e-Learning: Hanging-out in Digital Classrooms*

Research and Other Projects

- 2012-2015 Strategic Project with FTW Vienna, *Analysis on Semantically Structured Data from Smart Buildings and Smart Grids*, Technical Lead
- 2012-2016 Executive Assistant for Public Relations Projects of the Department of Computer Science at Johannes Kepler University Linz: *JKU Young Computer Scientists, You can make IT* (reporting directly to the Ministry of Science, Research and Economy)

Additional Training

- 2009 Executive Training in the Academy of the Austrian Red Cross
- 2007 Qualification as Trainer for First Aid Courses in the Curriculum of the Austrian Red Cross