# DynamoGraph: A Distributed System for Large-scale, Temporal Graph Processing, its Implementation and First Observations

Matthias Steinbauer
Institute of Telecooperation
Johannes Kepler University
Linz, Austria
matthias.steinbauer@jku.at

Gabriele Anderst-Kotsis
Institute of Telecooperation
Johannes Kepler University
Linz, Austria
gabriele.kotsis@jku.at

## ABSTRACT

Graph models have a long standing history as models for real world structures and processes. In recent research two important dimensions of graphs are described of particular importance. (1) Temporal aspects of graphs cannot be neglected for many current application scenarios such as social network analysis or the analysis of the global web graph. (2) The mentioned graph structures have grown to very large sizes such that traditional methodologies no longer hold. In this work a distributed computing framework designed for storing and processing of large-scale temporal graphs is presented. For this system a reference implementation in Java was created. In this paper first insight on the implementation and observations in using the system are discussed.

## Keywords

Distributed computing; Graph processing; Pregel; Temporal graph; Large-scale graph

## 1. INTRODUCTION

Graph and graph models have a long standing history in computer science. They serve as a model for real world structures in various disciplines. In the social sciences graphs are used to model social networks and sociometry is used to compute metrics over these models [14]. Biological processes such as protein-protein interaction can be modelled as a graph. Digital road maps can be represented as graphs, and finally the global web graph can be seen as a very large scale dynamic graph.

While there has been and still is continued research on graphs we see a rise of graph related research in computer science in the recent years. This is mainly due to two factors inherent in graph models that still leave room for further research. (1) In the past especially in applied computing the dimension of time has been neglected in graph models. Whereas real-world systems are subject to change over time. (2) Real-world systems can grow to very large scales such that processing paradigms and algorithms suitable for small graphs are not feasible. As discussed in section 2 both problems have been addressed. This paper argues that future systems for graph analytics must support storage and processing over large-scale, temporal graphs to provide better insight.

In the real world structures are dynamic i.e. social networks are changing over time, protein-protein interactions are in fact dynamic processes, digital road maps and of course also the global web graph are under constant change. Static views on these dynamic systems will show too much data and thus can blur metrics computed on static views. On the other hand for some applications the dynamics in a system might be of interest. Thus it is necessary that changes and their time of occurrence are saved alongside with the graph. Some graph metrics will even show new meaning when computed in the context of a temporal graph. Reachability measures might vary depending on the temporal context they are computed in.

Real world systems often also scale to very large sizes. Again the examples given above fall into this category. Systems modelled as graph will grow in two dimensions: (1) their diameter might grow as new vertices are added to the graph, and (2) the density of data will grow as new data is added in the temporal dimension. Very large graph models can easily grow to sizes where local memory of a single computer might not be sufficient anymore. Also computing resources of a single computer might prohibit that graph metrics over large data-sets can be computed in feasible time.

In this paper a distributed software system is proposed which provides its users with scalable storage mechanisms for large-scale temporal graphs alongside with a distributed processing framework that can be used by developers to implement distributed graph algorithms. The system aims to hide some of the complexities coming with distributed computing systems from software developers.

In this work the authors present related work from the fields of large-scale graph processing, and temporal graphs (see section 2). A framework for temporal graph representation and processing that can accommodate large-scale real world networks is discussed in section 3. A real-world reference implementation called DynamoGraph and some of its implementation details are presented in section 4. First observations of processing tasks executed on top of DynamoGraph are given in section 5 and section 6 concludes the paper.


*WWW'16 Companion,* April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4144-8/16/04.
http://dx.doi.org/10.1145/2872518.2889293.

## 2. RELATED WORK

The system presented in this paper is based on graph concepts which have been defined and discussed in great detail before. In general in graph theory the concept of dynamic graphs was defined. Dynamic graphs can be modelled in various different ways [8]. In this work we use the following definition: A dynamic graph is a pair $(V, E)$ where $V$ denotes the set of vertices and $E$ denotes the set of edges between any $v, u \in V$. A graph is called *vertex-dynamic* if the set $V$ changes over time and *edge-dynamic* if the set $E$ changes over time. Moreover a graph can be called *vertex-* and *edge-dynamic* if both sets change over time [4]. In a dynamic graph changes are not necessarily traceable, if a dynamic graph is observed at time $t$ one observes a static graph, its history is unknown.

Data structures that are able to preserve a graphs history are called temporal graphs. A temporal graph $T$ can be described as a set of graphs $T = \{G_1, G_2, G_3, \ldots, G_t\}$ where each $G_x \in T, G_x = (V_x, E_x)$ is called a static snapshot of $T$ at time $x$. Multiple such snapshots from $tm$ to $tn$ can be selected from $T$ we call the resulting $G_{tm\ldots tn}$ a timeframe snapshot of $T$. For temporal graphs new metrics (temporal proximity, temporal availability, etc.), and tactics for visualisation are required [7].

Large and temporal graphs have also spiked interest in applied science communities. Traditionally in applied sciences graph databases have always played an important role. Needless to say that graph databases still provide fast mechanisms for manipulating graph data and query mechanisms to retrieve graph data. Temporal aspects have been integrated in enterprise grade graph databases such as Neo4j [1]. In graph databases temporal aspects are often modelled as intermediate vertices representing timestamps. Which integrates well with these systems. However, graph databases usually are not optimized for temporal graph processing.

Several systems for temporal graph processing thus have been implemented in the past. From one thread of research the systems Chronos [3] and Immortalgraph [13] originate. Systems which try to optimize in-memory organization of temporal graphs to guarantee fast access times for processing tasks.

On the other hand there exist several approaches towards large-scale graph processing. Obviously many are discussed in the distributed computing communities. Many traditional graph algorithms use graphs represented in their adjacency matrix as their input. If these matrices are sparse and grow to very large sizes distributed matrix processing can provide a viable solution [5]. However, the adjacency matrix becomes impractical to handle if its cells contain complex data-structures as it would be required by temporal graphs.

GraphLab PowerGraph [9] tackles the size dimension with a parallel computing model which is designed for in-memory processing in distributed compute clusters. Its main aim was to create a graph based system that builds the foundation layer for machine learning tasks. PowerGraph is also the foundation of the commercially available Dato platform[1] and the in-memory graph processing system GraphChi[2].

For vertex-centric processing (representing vertices as documents) the Pregel [11] processing paradigm and extensions over it [16] have proven to be feasible in practice. For this processing paradigm an open source implementation (Apache Giraph[3]) exists that integrates well with state of the art Big Data platforms such as the Apache Hadoop[4] ecosystem. Similarly the Apache GraphX library provides distributed graph processing in the context of the Apache Spark project [21].

However, the mentioned systems are not designed for temporal graph processing and thus lack framework support for time-annotated data. In this work authors focus on the intersection of large-scale and temporal graphs for which currently no systems exist that support both data-management and processing tasks over such data.

## 3. PROCESSING FRAMEWORK

In the following the storage of temporal graph data in a distributed compute cluster (see section 3.1) and distributed processing over this data with extended Pregel concepts (see section 3.2) are discussed.

### 3.1 Temporal Maps

Temporal graphs have two major dimensions of growth. As more vertices get added to the graph the graph can grow in diameter, and as more time-stamped edges get added the graph grows in density. Partitioning of a temporal graph is the decision of whether to split data alongside its temporal or structural dimension. The problem of temporal partitioning is solved by storing individual snapshots of an individual graph on different computers. Often the diameter of the graph is the dominant dimension of growth such that even individual static snapshots of the complete graph cannot be hosted in memory of a single computer. Further processing resources of a compute cluster might not be facilitated optimally if the temporal graph is partitioned in the temporal dimension. This is why way was given to structural partitioning of the temporal graph in this particular work.

Structural graph partitioning is the task of distributing $N$ vertices over $H$ host computers. Each of these computers is responsible for data management and computation of its local graph partition $P_H$. In structural processing arcs between vertices can either connect vertices within a local partition or vertices in different partitions. The number of the arcs spanning between partitions are called the graph split (or cut). Since many graph algorithms i.e. reachability measures [20], PageRank [15], etc. use the arcs of a graph as guidelines to traverse the graph in structural graph partitioning algorithms aim to find minimal splits in the graph. In this work existing it is assumed that a partitioning function $p(v)$ exists that assigns a partition number to any given vertex $v$.

In a graph partitioned as described above it is possible to store each individual vertex of the graph as a single data structure or document. In practice the system presented in this paper stores data for individual vertices in-memory as maps and as JSON documents or other serialized representations in persistent memory. JSON representation is mainly used in scenarios where compatibility to other systems or manual investigation of individual vertices is required (debugging, data-manipulation through external tools). The arcs are stored within these vertex documents in lists con-

[1]Dato create intelligence: `https://dato.com`
[2]GraphChi: `https://github.com/GraphChi`
[3]Apache Giraph: `http://giraph.apache.org`
[4]Apache Hadoop: `https://hadoop.apache.org`

taining the arcs alongside with their attributes. To avoid loading of arc data from remote vertices and thus from remote hosts during processing each vertex has separate lists for incoming and outgoing arcs. This means that each arc is stored in the system twice.

The concept of a map was extended to what is named a temporal map. This is a map that can hold key-value pairs tagged with time-stamps. It is configured with a temporal resolution which defines the smallest unit of time that can be used in the map. Just as a regular map the temporal map defines a write operation $w(M, k, o, t)$ which writes the object $o$ with key $k$ into the map $M$. However the write operation is tagged with time-stamp $t$. A read operation over a temporal map $r(M, k, b, e)$ returns an object with the key $k$ from map $M$ which has a time-stamp $t$ that lies between the begin time $b$ and end time $e$. The presented behaviour leads to data conflicts in practice i.e. two different objects $o_1$ and $o_2$ were inserted with two different time-stamps. Real world implementations of a temporal map address this issue with conflict solving strategies such as returning the newer item by default, merging collections, and allowing developers to specify custom conflict solving strategies for certain keys.

In listing 1 the temporal representation of a vertex in monthly resolution is presented. Attributes were added to the temporal map at two different timestamps. From the listing it is clear that for each time-stamp a map can be stored which again can contain arbitrary attributes. In the example given, regular attributes and collections were added to the map.

**Listing 1: JSON document representing a temporal vertex**

```
{
  id: 39827736,
  resolution: 'MONTHS',
  '1420070400': {
    name: 'Rob Henderson',
    description: '',
    inEdges: [ {
      weight: 3.3,
      edgeType: 'PHONE',
      source: 39761932,
      target: 39827736, } ],
    outEdges: [ {
      weight: 4.0,
      edgeType: 'EMAIL',
      source: 39827736,
      target: 39761932, } ],
  },
  '1422748800': {
    inEdges: [ {
      weight: 4.0,
      edgeType: 'PHONE',
      source: 39761932,
      target: 39827736, } ],
    outEdges: [ {
      weight: 6.0,
      edgeType: 'EMAIL',
      source: 39827736,
      target: 39761932, } ],
  }
}
```

In the large-scale temporal graph framework presented in this paper each graph partition $P_H$ is a list of temporal maps $M$. Each map representing a single vertex in the temporal graph. Each host $H$ is responsible for data-management and computation in its local partition $P_H$.

## 3.2 Pregel Processing and its Extensions

The presented framework is inspired by Pregel [11] which was originally developed at Google Inc. In Pregel algorithms repeatedly execute a local compute function $c(v)$ in the context of each vertex $v$ in the graph. The compute function $c$ can only read and write the vertex $v$ and thus has no side-effects on any other vertex in the graph. Furthermore, since $c$ relies on local information only, $c$ can be executed in parallel for all vertices in the graph. From the computation performed in $c$ messages can be sent to any other vertex in the graph through the use of the message passing function $m(v_t, x)$ which sends a message $x$ to $v_t$.

As Pregel algorithms are repeatedly executing the compute function $c(v)$, these algorithms are iterative processes. Each iteration is called a *superstep*. Supersteps are numbered by the system global variable $t$. Messages sent through the messaging function in iteration $t$ can be processed in $c$ on the destination vertex in $t + 1$.

To allow the iterative Pregel process to run to a completion during each $c(v)$ run the algorithm can vote to halt the algorithm. A vote to halt will mark the vertex $v$ as inactive. The framework will not execute $c(v)$ for inactive vertices in consecutive iterations. Vertices can return from inactive to active state if they receive a message from another vertex. The global algorithm controller can halt the Pregel process if the number of active vertices reaches 0 meaning that all vertices have voted to halt.

As the original Pregel framework lacks some mechanisms to make it suitable for large-scale temporal graphs several extensions were made. First of all our Pregel-inspired framework supports temporal filtering. This means when the framework calls the compute function $c(v, \theta)$ only data from a certain filtering timespan $\theta$, determined by the developer when launching the algorithm, is visible in the data of $v$. Data is filtered in the framework, such that the results of potentially expensive merge operations can be cached in memory.

Furthermore the framework was extended by a global map $\Gamma$. This map can be used as global memory for algorithms. Framework users can use it to provide user-defined settings to algorithms (i.e. damping factor for a PageRank algorithm) and to retrieve results back from algorithm executions (vertex identifiers of the top-10 PageRank vertices). The compute function gets changed to $c(v, \theta, \Gamma)$. Inside of $c$ local processing is performed over a local copy of $\Gamma$. In between each iteration $t$ and $t+1$ the global map $\Gamma$ from all partitions $P_H$ gets merged according to user-defined merge strategies.

During algorithm setup an initialisation function $\kappa(\Gamma)$ is called. Developers can implement this function to generate the initial map $\Gamma$ with global parameters. Further, in-between each iteration the global function $\gamma(\Gamma)$ is called. This function can be implemented to manipulate the global map after each *superstep*. The extensions over Pregel and its inner workings are discussed in greater detail in related work: [16, 18].

In order to create algorithms in this Pregel-inspired framework, software developers need to provide implementations of the functions $c$, $\kappa$, and $\gamma$. The framework in turn is responsible for data-management, correctly executing the compute functions in the distributed compute cluster, properly routing messages, and monitoring the Pregel process. The support for $\kappa$, and $\gamma$ are extensions over previous versions of DynamoGraph (see [19] for initial proof-of-concept prototype, and [18] for more details and algorithms).

# 4. REFERENCE IMPLEMENTATION

In order to demonstrate the feasibility of the framework presented in 3, a reference implementation was created us-

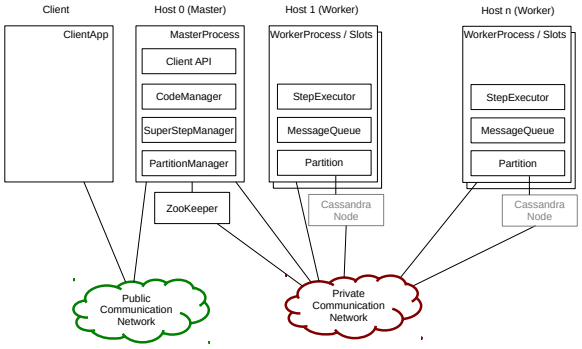


**Figure 1: Architecture of the DynamoGraph platform**

ing Java technology. The architectural overview of this reference implementation is presented in figure 1. In general it is a distributed computing architecture with hosts taking different roles. On the very left a *client* host is depicted, of which multiple instances can connect through a communication network to the host in *master* role. All the other hosts in the system are of *worker* role and are connected through a communication network with the *master*.

In typical setups on the *master* host also an instance of Apache ZooKeeper[5] service is running. While this service could also run on a dedicated hosts, and is in fact designed to run in ensembles of multiple distributed hosts. In smaller setups (such as in our test lab) the master host is not under heavy load such that ZooKeeper can run on the same machine. ZooKeeper is a distributed service which can provide configuration information, naming, and distributed synchronisation. It is used as a service directory within the system that holds relevant information such as IP-addresses and ports of individual DynamoGraph services on the individual hosts. Further, a standardized master-election scheme is used to determine which host is the master in the system. Finally, ZooKeepers ephemeral storage mechanism is used to monitor hosts for failure and to devise recovery strategies such as re-electing a new master and recovering lost data from backups.

The *master* node is responsible for data management and partition management (all implemented in the *PartitionManager* component). The *PartitionManager* selects a partitioning function (which can be provided by the framework users) $p(v)$ and wraps it in a Java object called *PartitionTable*. This partition table is distributed to all worker nodes in regular intervals to allow the workers to perform partition lookups locally. In the simplest case the *PartitionTable* is implemented as mathematical function such as a modulo operation over the vertex identifier and the number of partitions.

Further the *master* node hosts a code repository called *CodeManager* which is responsible for importing and registering user provided code. Framework users can submit their user defined code (implementations of the functions $c$, $\gamma$, and $\kappa$) as implementation of the abstract class *SuperStep* bundled in a JAR file. These JAR files get unpacked, verified, and loaded to all *worker* nodes as per instructions of the *CodeManager*.

In the component *SuperStepManager* actually executing code is administered. When client code schedules a superstep execution in the cluster, the instruction to execute code gets registered with this component and the *master*, through message passing mechanisms, instructs all *worker* nodes to execute the individual phases of the superstep (execution of $c$ for all active vertices, and message routing) in their local partition. The *master* node awaits feedback from the individual partitions and executes $\kappa$ (if required) before scheduling further iterations or halting the algorithm.

On the *worker* role hosts in real-world installations multiple instances of the worker process, named *slot*, are executed in parallel. This is to facilitate all the processors installed in the *worker* nodes. Each *slot* is responsible for a single partition. In an individual *slot* the component *Parititon* is responsible for data-management in the local data-partition. Data manipulation jobs are executed within this component.

The *StepExecutor* component inside of the *slots* is responsible for locally executing the superstep function $c$ for each active vertex in the graph. In parallel to executing $c$ the slots start with message routing. Messages stored in the local message queues (component *MessageQueue*) are bundled up in larger packages and routed to their destination partitions. After a *slot* has completed $c$ for each active vertex in its partition it waits for the *MessageQueue* to complete routing and then reports back successful step execution alongside with the number of active vertices in the partition to the *SuperStepManager* at the *master*.

## 4.1 Extended Requirements

A real-world implementation of such a framework poses further requirements. For DynamoGraph mainly the requirements of multi-tenancy, fault-tolerance, and persistency were identified. For real-world application scenarios users of DynamoGraph will obviously not be willing to setup a compute cluster for just a single analysis task. It is far more likely that multiple different datasets or even versions of the same dataset are subject to be analysed in a certain project. This means that DynamoGraph must support multi-tenancy in such a way that multiple temporal graphs can be hosted on the same cluster system. In DynamoGraph this is implemented by namespaces. Just like tablespaces segregate different database schemas, namespaces segregate different temporal graphs. Data manipulation operations and algorithm execution are always executed in the context of a single namespace.

It is well known that distributed systems have a higher probability of failing. Especially in the cloud-computing arena systems must be built in a fashion that individual components of larger systems can fail. In DynamoGraph failure of a single compute node could mean loss of multiple graph partitions. To counteract DynamoGraph clusters can run in a fault-tolerant mode where for each partition a configurable number of backup partitions are maintained. All data manipulation operations executed in the context of any vertex are repeated in the backup partition. Currently the system only supports an eventual consistency model. In case of node-failure all running algorithms are stopped, the *master* process computes a new partition table and instructs all partitions (also the backup partitions) to apply the new partition schema.

Finally, with large dataset sizes that get loaded for real-world analysis tasks the upload of datasets takes significant time and failure or the need to restart the complete cluster

[5]Apache ZooKeeper: `https://zookeeper.apache.org`

will lead to data loss. Thus DynamoGraph not only supports graph models to be kept in memory (volatile mode) but also provides a persistence backend. Persistence modules can be added to the system through a plugin infrastructure with implementations for JSON files and the Cassandra[6] key-value store being provided by the framework. In the case the Cassandra persistence backend is used, each *worker* node also runs a node of a Cassandra ring (see figure 1). In this case fault-tolerance is delegated to Cassandra.

## 4.2 Exemplary Algorithm

To illustrate how prospective developers can use the DynamoGraph framework the following section will discuss the implementation of an exemplary algorithm in greater detail. As an example the PageRank citation ranking algorithm [15] was implemented for DynamoGraph. The algorithm was chosen because it itself is an iterative process. Initially Pregel style processing was developed exactly for such iterative processes.

In the PageRank algorithm first all vertices in the graph get initialized with an initial rank value $r_0$. After initialization the compute function $c$ performs the following steps: (1) Divide the current rank (optionally multiplied by a damping factor) of vertex $v$ by the number of outgoing arcs of $v$. (2) Use the messaging function $m$ to send the computed value to all neighbouring vertices connected through outgoing arcs. In the next iteration each vertex sums up the received PageRank values and sets the result as their new PageRank, then the process continues as described above.

Halting of the algorithm can be implemented through different criteria. Either PageRank is computed for a fixed amount of iterations which will give sufficient results for many applications i.e. finding the top ranked vertices in the graph or a constant swinging threshold $\phi$ is used. If a vertices PageRank is changed by less than $\phi$ instead of continuing processing the vertex simply votes to halt. The vertices' PageRanks will converge towards an optimum with an accepted error of $\phi$.

**Listing 2: PageRank implemented for DynamoGraph**

```
1  public void execute(
2    List<VertexMsg> messages, VertexContext vertexContext,
3    SuperStepContext superstepContext,
4    Timeframe timeframe, Vertex vertex) {
5    if(this.getStep() >= PageRank.MAX_ITER) {
6      voteToHalt(vertexContext); return;
7    }
8    if(this.getStep() == 0L) {
9      setPageRank(vertexContext, PageRank.INITRANK);
10     Collection<Edge> outEdges = vertex.getWeightedOutEdgesReading();
11     float outRank = (pageRank * PageRank.DAMP) / outEdges.size();
12     for(Edge out : outEdges) {
13       sendMessage(out.getTarget(), outRank);
14     }
15   }else{
16     if(messages.size() > 0) {
17       float changedby = 0.0f; float sumIncoming = 0.0f;
18       for(VertexMessage m : messages) {
19         sumIncoming += m.getFloatValue();
20       }
21       changedby = setPageRank(vertexContext, sumIncoming);
22       if(changedby >= PageRank.SWING_THRESHOLD) {
23         Collection<Edge> outEdges = vertex.getWeightedOutEdgesReading();
24         float pageRank = getPageRank(vertexContext);
25         float outRank = (pageRank * PageRank.DAMP) / outEdges.size();
26         for(Edge out : outEdges) {
27           sendMessage(out.getTarget(), outRank);
28         }
29       }
30     }
31     voteToHalt(vertexContext);
32   }
33 }
```

[6]Cassandra: `https://cassandra.apache.org`

In listing 2 a Java source code listing of the compute function $c$ of PageRank is given. In practice $c$ is implemented by overriding the method `execute` of the abstract class `SuperStep`. In the listing the first parameter (`messages`) refers to a list of incoming messages, `vertexContext` is a map for volatile vertex-local storage, `superstepContext` refers to the global map $\Gamma$, `timeframe` denotes the filtering timeframe $\theta$, and finally the parameter `vertex` describes the vertex $v$ which is currently processed.

The presented implementation performs the initialization steps in iteration $t = 0$ (code lines 9 to 14). Regular iterations of computing PageRank and distributing updated values to neighboring vertices are implemented in lines 16 to 31. A halt condition with a swinging threshold is implemented in the if-block starting from line 22. To avoid non-halting algorithms a backup halt condition, which stops the algorithm after a developer defined maximum of iterations, is implemented at line 5.

Developers can wrap up code with all relevant dependencies and a manifest file into a JAR file. Using the DynamoGraph Java API it is possible to upload such JAR files to the compute cluster and instruct the cluster to execute the algorithm in the context of a namespace and in the context of a certain timeframe present in the data. In listing 3 the necessary steps are presented, in the example it is assumed that the variable `service` is a properly configured and connected DynamoGraph Java API instance. In lines 1 to 6 the file `pagerank.jar` gets loaded to a byte array, uploaded to the cluster, and the code gets registered with its class name `at.jku.tk.dynamo.step.PageRank`. In lines 8 through 10 the uploaded code gets executed in the cluster. On line 8 a global map is generated, however, not further initialized. Line 9 shows the definition of a timeframe object which restricts the temporal dimension of the processed data.

It is also possible to query the status of currently executing jobs. Lines 12 to 18 show how a developer can make her local process wait for a currently running job. In the case the algorithm ran to to completion the global memory with results can be retrieved from the cluster. In the error case it is possible to query the exceptions that lead to the algorithm failing.

**Listing 3: Code Upload and Execution**

```
1  File testFile = new File("pagerank.jar");
2  FileInputStream in = new FileInputStream(testFile);
3  ByteArrayOutputStream bout = new ByteArrayOutputStream();
4  IOUtils.copy(in, bout);
5  String clasz = "at.jku.tk.dynamo.step.PageRank";
6  service.loadCode(clasz, bout.toByteArray());
7
8  SuperStepContext context = new SuperStepContext();
9  TimeFrame timeframe = new TimeFrame(1420070400L, 1422748800L);
10 long id = service.executeAlgorithm("testgraph",
11   "at.jku.tk.dynamo.step.PageRank", context, timeframe);
12 if(service.waitForCompletion(id) == ExecutionProfileState.Completed) {;
13   context = service.queryAlgorithmContext(id);
14 }else{
15   System.err.println("Step failed!!! " + service.queryAlgorithmState(id));
16   Exception e = service.queryExceptions(id);
17   e.printStackTrace();
18 }
```

# 5. OBSERVATIONS

DynamoGraph as a software stack has now matured enough to be usable for first test-drives. In general the algorithm execution scales in that sense that a larger number of processors can be used to compute faster or to process over larger problem sets. This has already been shown in previous work, on a work-in-progress prototype of DynamoGraph [18].

In current research efforts DynamoGraph is analyzed in greater detail. Datasets of different size are uploaded to the cluster, first experiments with visualization and temporal graph algorithms are conducted. Currently data from three different sources are available.

Firstly, data from the Enron dataset [6] which was made available to the scientific community during the Enron scandal was uploaded see also [18]. This dataset is of medium size, it contains 200,399 email messages (edges) belonging to 158 users with an average of 757 messages per user. While the dataset can still be easily analysed on a single computer using traditional methods it is already large enough to see performance benefits from running analysis tasks in parallel and in a distributed system.

Consequently further datasets forming temporal graphs were obtained and imported. Especially the MIT Reality Commons[7] project has proven to be a valuable source of temporal graph data. Importers for the Reality Mining [2] and Social Evolution [10] datasets were created. While these datasets are rather small in diameter (80 to 100 vertices) they contain dense temporal information over longer periods of time. Thus these datasets have proven to be valuable to test implementations of distributed algorithms.

Finally, the data available in the Click Dataset [12] was obtained from the Center for Complex Networks and Systems Research group at Indiana University Bloomington. This data contains anonymized and condensed HTTP header information from incoming and outgoing web traffic of the Indiana University Bloomington for the time period from Sept 2006 through May 2010. This amounts to 53.5 billion HTTP requests that are available as a compressed and encrypted dataset of 2.5 TB size. This data is currently imported to DynamoGraph with plans to run temporal PageRank in sliding time-windows of varying size (3 to 6 month).

# 6. CONCLUSIONS

In this paper the DynamoGraph framework for distributed data-management and processing over large-scale temporal graphs was presented. It was highlighted how potential framework users (software developers, researchers, etc.) are able to utilize the system to run data analysis tasks over potentially large datasets. From code examples it was made clear, that the complexities of distributed computing (message passing, synchronization, etc.) are mostly hidden from software developers. It was elaborated how real world datasets of different sizes are currently made available in the framework such that the system can be tested and its performance being evaluated in depth in future work. Work-in-progress projects, that utilize DynamoGraph, and their application scenarios are discussed in greater detail in [17].

# 7. REFERENCES


[1] C. Cattuto, M. Quaggiotto, A. Panisson, and A. Averbuch. Time-Varying Social Networks in a Graph Database: a Neo4j Use Case. In *GRADES*, New York, USA, June 2013. ACM.

[2] N. Eagle and A. Pentland. Reality Mining: Sensing Complex Social Systems. *Personal and Ubiquitous Computing*, 2006.


[7]MIT Reality Commons: `http://realitycommons.media.mit.edu/`


[3] 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.

[4] F. Harary and G. Gupta. Dynamic Graph Models. *Mathl. Comput. Modelling*, 25(7):79–87, 1997.

[5] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. In *ICDM*, 2009.

[6] B. Klimt and Y. Yang. Introducing the Enron Corpus. Technical report, Language Technology Institute, Carnegie Mellon University, 2009.

[7] V. Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.

[8] F. Kuhn and R. Oshman. Dynamic networks. *ACM SIGACT News*, 42(1):82, Mar. 2011.

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *UAI*, July 2010.

[10] 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.

[11] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD*, 2010.

[12] M. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani. Ranking Web Sites with Real User Traffic. In *WSDM*, 2008.

[13] 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.

[14] M. D. Moreno. *Who Shall Survive?* 2nd edition, 1953.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

[16] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*. ACM, July 2013.

[17] M. Steinbauer and G. Anderst-Kotsis. Using DynamoGraph: Application Scenarios for Large-scale Temporal Graph Processing. In *iiWAS*, Brussels, Belgium, 2015.

[18] M. Steinbauer and G. Anderst-Kotsis. DynamoGraph: Extending the Pregel Paradigm for Large-scale Temporal Graph Processing. *IJGUC*, to appear 2016.

[19] M. Steinbauer and G. Kotsis. Towards Cloud-based Distributed Scaleable Processing over Large-scale Temporal Graphs. In *WETICE*, 2014.

[20] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Characterising temporal distance and reachability in mobile and online social networks. In *SIGCOMM*. ACM, Jan. 2010.

[21] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. GRADES '13, New York, USA, 2013. ACM.