Time-varying Social Networks in a Graph Database

[A Neo4j Use Case]

Ciro Cattuto
Data Science Lab
ISI Foundation
Torino, Italy

André Panisson
Data Science Lab
ISI Foundation
Torino, Italy

Marco Quaggiotto
Data Science Lab
ISI Foundation
Torino, Italy

Alex Averbuch
Neo Technology
Stockholm, Sweden

ABSTRACT

Representing and efficiently querying time-varying social network data is a central challenge that needs to be addressed in order to support a variety of emerging applications that leverage high-resolution records of human activities and interactions from mobile devices and wearable sensors. In order to support the needs of specific applications, as well as general tasks related to data curation, cleaning, linking, post-processing, and data analysis, data models and data stores are needed that afford efficient and scalable querying of the data. In particular, it is important to design solutions that allow rich queries that simultaneously involve the topology of the social network, temporal information on the presence and interactions of individual nodes, and node metadata. Here we introduce a data model for time-varying social network data that can be represented as a property graph in the Neo4j graph database. We use time-varying social network data collected by using wearable sensors and study the performance of real-world queries, pointing to strengths, weaknesses and challenges of the proposed approach.

Categories and Subject Descriptors

E.2 [**Data**]: Data Storage Representations—*linked representations*; H.2.1 [**Information Systems**]: Database Management - Logical Design—*data models, schema*; J.4 [**Computer Applications**]: Social and behavioral Sciences—*sociology*

General Terms

Graph Database, Social Networks, Wearable Sensors

Keywords

social networks, temporal networks, graph databases, neo4j

1. BACKGROUND

A growing variety of applications produce an increasing quantity of information about the social behaviour of their users. People leave behind a large amount of digital traces about their daily activities and applications that take advantage of these resources have the potential to change the way people live their lives. Mobile devices and wearable sensors with various sensing technoare able to collect information about social interactions between people. In order to preserve the temporal information of such interactions, collected data can be represented as time-varying social graphs, where nodes represent individuals, edges represent interactions between them, and both the graph structure and the attributes of nodes and edges change over time. In this model, both nodes and edges can have rich attributes and are associated to a temporal structure.

Data formats for exchanging time-dependent graphs are available, see for instance the GEXF format [1]. Efficiently mining large time-varying graphs, however, requires a database and a representation that can support complex topological queries, temporal queries, multi-scale temporal indexing and aggregation, and more. A growing research community working on temporal networks [11] may benefit from sound and efficient techniques to represent, store and query dynamic graphs.

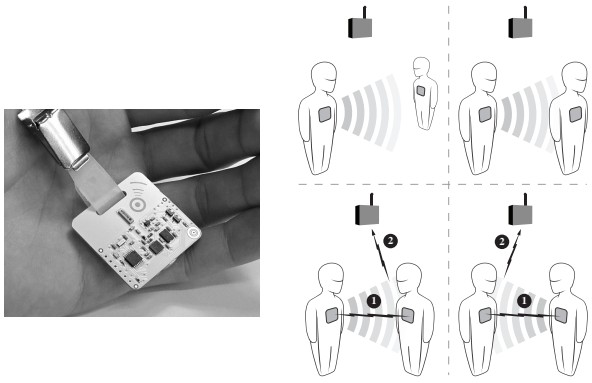
2. SOCIAL NETWORKS FROM WEARABLE SENSORS

In the following we use time-resolved behavioral social networks measured by the SocioPatterns collaboration[1] [10] by using wearable proximity sensors. The SocioPatterns collaboration aims at building a high-resolution atlas of human contact in a variety of indoor social environments, to be used for research on human mobility, computational epidemiology, opportunistic networks, and related domains.

Participants in the use cases are asked to wear badges equipped with active Radio Frequency Identification (RFID) devices (Figure 1(a)) that are programmed to sense and report close-range proximity relations between devices. The wear-

[1] `http://www.sociopatterns.org`

able devices engage in bi-directional radio communication, and the exchange of low-power radio packets is used as a proxy for the face-to-face proximity of participants, as illustrated in Figure 1(b). The spatial range for proximity detection can be tuned from several meters down to face-to-face proximity by varying the radio power of the proximity-sensing packets. At the highest spatial resolution, the exchange of radio packets is only possible when two persons are at close range ($\sim$ 1–1.5m) and facing each other, since the human body acts as a RF shield at the carrier frequency used for communication. The operating parameters of the devices were chosen so that face-to-face proximity relations can be assessed with a probability in excess of 99 % over an interval of 20 seconds, which is a fine enough temporal scale to resolve human mobility and proximity relations at social gatherings. The sensed proximity relations (or *contacts*) between individuals are received by RFID readers installed in the environments and relayed to a centralized data collection system for post-processing, storage and subsequent analysis. Once a contact has been detected, it is considered ongoing as long as the involved devices continue to exchange at least one radio packet for every successive interval of 20 seconds. Conversely, a contact is considered terminated if an interval of 20 seconds elapses with no packet exchange.



(a) (b)

Figure 1: (a) Wearable proximity sensor used by the SocioPatterns collaboration to mine close-range and face-to-face proximity in a variety of real-world settings. (b) Proximity sensing strategy. The wearable sensors engage in bidirectional ultra-low power radio communication (1). Packet exchange is only possible when two sensors are sufficiently close in space. At the lowest power used to sense proximity, packet exchange is only possible when the individuals wearing them are at close range (1-1.5m) and face each other (bottom panels). The sensed proximity relations are periodically relayed at higher power (2) to a centralized data collection system.

The proximity-sensing platform described above was deployed in several different settings, yielding data on time-resolved human proximity in conferences, hospitals, schools and museums. The specific dataset we use for the present study was collected at the 20$^{\text{th}}$ ACM Hypertext 2009 conference [12], from June 29th to July 1st 2009, and is available to the public [3]. Participants were invited to provide online social network data linked to their wearable sensor and were offered a service, Live Social Semantics [8], that allowed them to browse their social neighborhood, their encounters at the conference, and discover shared contacts and interests by mining the sensed social graph as well as the linked online social networks and user metadata.

The data we use provides, for each pair of participants, the detailed sequence of their contacts, with beginning and ending times. These data can be represented as time-varying proximity networks, obtained by temporally aggregating the raw data stream from the proximity sensors over temporal *frames* of a given duration, here chosen as $\Delta t = 20$s. For each consecutive time interval (frame) of duration Δt we build a *proximity graph* where nodes represent individuals, and edges represent proximity relations between individuals that were recorded during the corresponding frame. Within a frame, an interaction is considered active from the beginning of the frame to the end of the frame. In this representation, interactions and actors appear or disappear at frame boundaries only.

3. TECHNICAL CHALLENGES

Representing and efficiently querying time-varying social network data is a problem that needs to be tackled in order to support a variety of important tasks, including data curation, cleaning, linking, batch post-processing, and analysis. Solving this problem requires several challenges to be overcome. In particular, those related to data modeling and to data storage and retrieval.

3.1 Modeling Time-Varying Networks

Whereas simple representations based on (time-varying) adjacency matrices or adjacency lists allow to build specialized data processing pipelines, they have numerous limitations. They lack the flexibility required to harmonize the previously mentioned tasks, face scalability issues with large datasets, and provide constrained semantics for querying and exposing data to services and applications (e.g., for visualization). To this end, it is crucial to design models for time-varying social network data that can be efficiently deployed in databases, and to allow rich queries involving combinations of social network topology, temporal information, and metadata.

3.2 Storage & Retrieval of Large Networks

Querying network/graph data in a performant manner is difficult, even more so when the graph dataset is large and persistent. This difficulty stems from specific characteristics of graph data and graph access patterns:

- The topology of real-world graphs is heterogeneous. Graph datasets from systems such as social networks, online and offline infrastructural networks, records of human-driven activity and a host of natural phenomena exhibit fat-tailed statistics for node connectivity [13]. This results in the inevitable high heterogeneity of node connectivity, and the "dense node" problem, where a small fraction of nodes have connections to a large fraction of other nodes. Dense nodes are not only expensive to process when encountered, their high con-

nectivity also means they are encountered with great frequency.

In our data there are two main sources of heterogeneity: the first is the topological heterogeneity of the social networks we measure, the second is the temporal heterogeneity in activity due to the bursty nature of human dynamics [9, 15, 16]. These heterogeneities are reflected in the properties of the dynamic social graphs we want to model in Neo4j.

- Graph problems are data driven, that is, computation is largely dictated by graph topology. As most graphs are very heterogeneous, it is difficult to know the computation structure of graph algorithm in advance. Consequently, optimizing the execution of graph algorithms is a non-trivial problem.
- Due to their heterogeneity, graph access patterns typically have poor spatial memory locality. This results in large amounts of random memory access.
- Graph algorithms exhibit high data access to computation ratio - the runtime of most graph algorithms is dominated by memory access.

The SocioPatterns use case requires a graph storage and retrieval technology that supports: a rich graph data model, reliable storage of large graphs, and efficient execution of complex queries on large heterogeneous graphs. The Neo4j [6] graph database fulfills these requirements, and was therefore chosen for this work.

In general, graph databases are very well suited to this use case, but Neo4j was deemed particularly fitting due to the following reasons: support for the property graph data model [14]; persistent, transactional storage of very large graphs; support for deep graph analytics via efficient many-hop traversals; and support for the Cypher [4] declarative graph query language.

4. DATA MODEL

We assume that the social network is measured over adjacent discrete intervals of time, henceforth referred to as *frames*. A frame is the finest unit of temporal aggregation used and is associated with a time interval defined by start time and end time. A frame allows to retrieve the status of the social network during the corresponding time interval, it is thus associated with a social graph at a given point in time.

The nodes of such social graphs represent individuals and edges represent, for instance, the proximity relations of individuals during the time interval of the frame. Though the model presented here considers edges as undirected and weighted, it generalizes to directed weighted edges and to multi-relational networks.

4.1 Time-Varying Social Network

The following is a data model for time-varying social networks, used to represent the SocioPatterns data in the Neo4j graph database.

Modeling a specific domain as a graph often requires a more complex model than the obvious choice of representing domain objects as nodes and the relationships between them as edges. In general, it is necessary to define: a domain graph schema, which models domain level entities and the relationships between them; a data graph schema, the underlying graph data model; and a translation between these two abstractions.

This can be a complex task, all the more when the domain works with dynamically mutating, time-dependent graphs. Representing such graphs requires that domain-level entities can be associated with specific time intervals, and that the model supports execution of temporal queries. Relationships between entities and the time intervals with which they associate are first modeled at the domain graph level. Therefore, any single domain level entity is likely to be represented by multiple nodes and edges in the underlying data graph - nodes and edges in the application domain are represented by subgraphs in the underlying property graph.

In our application individuals of the social graph are represented by their wearable sensor. To avoid ambiguity and stay close to the language of our application, the following introduces all terms used to represent entities in the time-varying social graph:

- A time-dependent graph as a whole is accessed as a `RUN` node, corresponding to a time-resolved record of social network evolution. A `RUN` node is connected to the reference node by means of a `HAS_RUN` relation.
- `RUN` nodes have `RUN_FRAME` relations to all the `FRAME` nodes. They also have a `RUN_FRAME_FIRST` relation to the first frame of the graph history, and a `RUN_FRAME_LAST` to the last one.
- Each `FRAME` node points to the successive frame by means of a `FRAME_NEXT` relation.
- Each `FRAME` node points to the status of the social graph during the corresponding time interval: it has `FRAME_ACTOR` relations to `ACTOR` nodes (representing individuals wearing sensors) and `FRAME_INTERACTION` relations to `INTERACTION` nodes (representing a proximity/contact relation). Timestamps and interval metadata are represented as attributes of the `FRAME` node.
- An `INTERACTION` node has two `INTERACTION_ACTOR` relations to the `ACTOR` nodes that the interaction connects to one another.
- Time-dependent attributes for an edge of the social graph (for example, time-dependent edge weights[2]) are represented as attributes of the `FRAME_INTERACTION` relations that associate that interaction with the corresponding frame(s).
- Similarly, time-dependent attributes of an `ACTOR` node in the social graph are represented as attributes of the `FRAME_ACTOR` relations that associate that node with the corresponding `FRAME`.

[2]Edge weights were calculated in different ways. Often simply as the number of frames in which an interaction is present, sometimes in more complex ways.

- All `ACTOR` and `INTERACTION` nodes are connected to the main `RUN` node with `RUN_ACTOR` and `RUN_INTERACTION` relations.

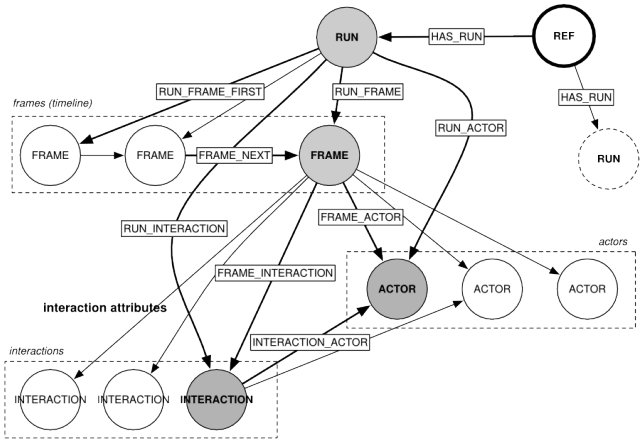


Figure 2: Time-varying Social Network Data Model

4.2 Temporal indexing

Although graph evolution can be tracked by walking from one frame to the next, efficient random access to the frame timeline can be better supported by suitably indexing the time-stamped sequence of `FRAME` nodes. This can be achieved in different ways, such as using binary trees (as in the Neo4j Timeline class [2]), or attaching to the `FRAME` nodes temporal attributes that can be indexed, or mirroring the natural temporal hierarchy of the data (year/month/day/hour/...) with hierarchical temporal relations between nodes.

Here we choose the latter technique, as it preserves the natural units of temporal aggregation in our data (days, hours, etc.). The temporal indexing structure we use[3] (shown in Figure 4.2) is similar to the multilevel indexing structure proposed in the Cypher cookbook [5].

- We build a tree that explicitly represents the temporal hierarchy of the dataset. The nodes of the tree are `TIMELINE` nodes. The top-level node, which is the entry point for the temporal index, is reachable from the `RUN` node through a `HAS_TIMELINE` relation.
- Nodes at each level of the tree have `NEXT_LEVEL` relations to nodes at the level below.
- The nodes at each level of the tree represent different scales of temporal aggregation, according to the time units that are most appropriate for the dataset at hand. For simplicity, the tree in Figure 4.2 has only day and hour levels, but in may be extended to have a deeper hierarchy, such as years, months, days, hours, minutes, and so on.
- At each level, time attributes are associated with the `NEXT_LEVEL` relations

[3]The selected multilevel index can co-exist with other indexing structures, its use does not prevent us from using different, additional indexes in future

- The nodes at the bottom level of the tree correspond to the finest scale of temporal aggregation (hours, in figure) and are connected to the indexed `FRAME` nodes via `TIMELINE_INSTANCE` relations. The timestamp of each `FRAME` node is replicated as a relation attribute of the corresponding `TIMELINE_INSTANCE` relation.

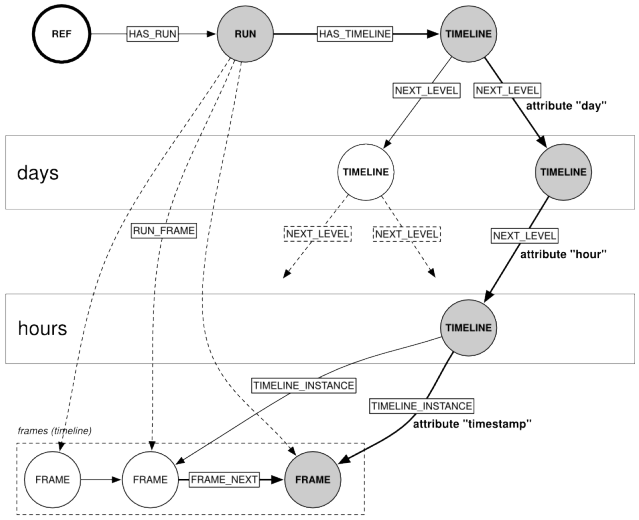


Figure 3: *TimeLine* Temporal Index

5. QUERYING TIME-VARYING NETWORKS

The SocioPatterns collaboration datasets provide realistic use cases, with topological and temporal hetereogeneities that would be expected in a real time-varying network. We provide a testing scenario using the empirical dynamic social network [3] measured at the ACM Hypertext 2009 conference by the SocioPatterns collaboration. The network of proximity relations among the conference attendees was recorded every 20 seconds for approximately three days. In this dataset, nodes are individuals and edges represent face-to-face proximity relations of individuals.

The test dataset is available as a dynamic GEXF document, and to simplify testing with the proposed data structures, a simple Python script was developed to load the GEXF document into a Neo4j store, via its REST interface. It contains data for total duration of slightly more than three days and is relatively small: 113 persons (`ACTOR` nodes), 2163 proximity relations (`INTERACTION` nodes), 13956 frames of 20 seconds each (`FRAME` nodes). Production datasets are typically much larger than this, involving 100-1,000 persons, 50,000-100,000 proximity relations, and 70,000-100,000 frames.

5.1 Test Queries

In the following a number of sample queries are provided, which executed against a ACM Hypertext 2009 conference dataset. The reported timings for each query are obtained by executing 30 runs of the same query. Table 1 shows the median time, the first 5% quantile and the last 95% quantile of the execution timings.

Q1. Get all time frames of run "HT2009", recorded between 9:00-13:00 of July 1st 2009, ordered by timestamp

```
START root = node(root_node_id)
MATCH root-[:HAS_RUN]->run-[:HAS_TIMELINE]->tl,
    tl-[y:NEXT_LEVEL]->()-[m:NEXT_LEVEL]->month,
    month-[d:NEXT_LEVEL]->[h:NEXT_LEVEL]->hour,
    hour-[:TIMELINE_INSTANCE]->frame
WHERE run.name="HT2009" and y.year=2009 and m.month=7
    and d.day=1 and h.hour>=9 and h.hour<13
RETURN frame ORDER BY frame.timestamp
```

Q2. Get the names of all persons present in a given frame

```
START frame = node(some_frame_id)
MATCH frame-[:FRAME_ACTOR]-actor
RETURN actor.name}
```

Q3. Get the weighted proximity graph during a given frame, filtering out the weak contacts

```
START frame=node(some_frame_id)
MATCH frame-[r:FRAME_INTERACTION]-int
WHERE r.weight > 20
RETURN int.actor1, int.actor2, r.weight
```

Q4. Get a list of all persons, and for each person get the number of frames in which they were present

```
START run = node(some_run_id)
MATCH run-[:RUN_ACTOR]->actor<-[r:FRAME_ACTOR]-()
RETURN actor.name, count(r)
```

Q5. Get the names of all persons that were present in more than 1000 frames, ranked by time of presence

```
START run = node(some_run_id)
MATCH run-[:RUN_ACTOR]->actor<-[r:FRAME_ACTOR]-()
WITH actor.name as name, COUNT(r) as freq
WHERE freq > 1000
RETURN name, freq ORDER BY freq DESC
```

Q6. List all distinct days on which an actor was present

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

Q7. Return the names of all persons that were in the proximity of a given user, sorted alphabetically

```
START actor1 = node(some_actor_id)
MATCH actor1<-[:INTERACTION_ACTOR]-interaction,
    interaction-[:INTERACTION_ACTOR]->actor2
RETURN actor2.name ORDER BY actor2.name
```

Q8. Return the names of all persons that were in proximity of a given user on the first day of the experiment (June 29)

```
START actor1 = node(some_actor_id)
MATCH actor1<-[:INTERACTION_ACTOR]-int,
    int-[:INTERACTION_ACTOR]->actor2
WITH int, actor2
MATCH ()-[d:NEXT_LEVEL]->()-[:NEXT_LEVEL]->tl,
    tl-[:TIMELINE_INSTANCE]-()-[:FRAME_INTERACTION]-int
WHERE d.day = 29
RETURN DISTINCT(actor2.name)
```

Query	Median	5% - 95% quantiles
Q1	83ms	72ms - 89ms
Q2	2ms	1ms - 4ms
Q3	2ms	1ms - 2ms
Q4	2313ms	2265ms - 2362ms
Q5	2315ms	2253ms - 2354ms
Q6	71ms	70ms - 75ms
Q7	4ms	3ms - 5ms
Q8	45ms	44ms - 61ms
Q9	18ms	10ms - 27ms
Q10	32ms	32ms - 47ms

Table 1: For each query described, we show the median value, the first 5% quantile and the last 95% quantile of the execution timings, collected by executing 30 runs of the query.

Q9. Find all the common neighbors of any two users

```
START actor1 = node(actor1_id), actor2 = node(actor2_id)
MATCH actor1<-[:INTERACTION_ACTOR]-interaction,
    interaction-[:INTERACTION_ACTOR]->actor
WITH COLLECT(actor) as neighs1, actor2
MATCH actor2<-[:INTERACTION_ACTOR]-interaction,
    interaction-[:INTERACTION_ACTOR]->actor
WHERE actor IN neighs1
RETURN actor
```

Q10. Compute the degree of all persons in the contact graph

```
START run = node(some_run_id)
MATCH run-[:RUN_ACTOR]-actor-[r:INTERACTION_ACTOR]-()
RETURN actor.name, COUNT(r) ORDER BY COUNT(r) DESC
```

6. CONCLUDING REMARKS

Our goal is to start discussion about best practices for modeling time-dependent graphs in graph databases. We encourage critique of our models, further testing using our data and/or code [7], and general suggestions related to the domain. The following is a summary of our observations, open questions, and avenues for future work.

6.1 Performance

Overall, the combination of our chosen data models (Section 4) and Neo4j proved to perform well when querying the sample dataset, performing exploratory data analysis, and research-oriented data mining. The simple experiment results reported are in no way a performance benchmark, nor do they reflect the general performance of Neo4j. However, they were valuable when tuning the performance of our application in its target domain, and have proved useful for identifying key performance bottlenecks - these are discussed below.

Most performance issues encountered were rooted in the same general cause: densely connected nodes. As highlighted in Section 3.2, the difficulty of processing graphs that contain densely connected nodes is known, by academia and industry at large, and by Neo Technology in particular. Neo Technology are aware of the requirement to process such graphs and have informed us they are working on improvements to Neo4j that directly address these issues.

Pre-computing Expensive Operations

There is great value in being able to run queries concurrently with data ingestion. One advantage graph databases provide is making this possible in the majority of cases. However, certain scenarios make it difficult to do so in a computationally efficient manner. As a remedy to improve performance, expensive or frequent operations can be pre-computed during a post-processing step (between data ingestion and query phases), in effect enriching the data model. For example, to solve performance issues that were encountered when executing QUERY 4 and QUERY 5 (Section 5.1), we modeled the number of `FRAMES` an `ACTOR` is connected to as a property of the `ACTOR` node - decorating the node with additional metadata. This alleviated the dense node-related performance bottleneck, and greatly improved performance.

Identifying Densely Connected Nodes

It was found that QUERY 6 (Section 5.1) scaled poorly with the size and density of the dataset. When testing with a 10-day dataset covering 200 `ACTORS`, we observed each `FRAME` typically had more than 20,000 relations, and at this stage execution time grew to the point that made it impractical to execute the query interactively.

To understand the problem consider a real-world example, a museum with 1000 daily visitors (`ACTORS`). Assume each `FRAME` represents a day, and people (`ACTORS`) change continuously, but the venue always operates at capacity. The time interval of a `FRAME` associates with a high number of `ACTORS` and `INTERACTIONS`. With 1000 `ACTORS`, there can in theory be up to 500,000 `INTERACTIONS` (a clique), and in practice between 20,000-50,000. This would result in `FRAMES` with 1000 `FRAME_ACTOR` relationships and 20,000-50,000 `FRAME_INTERACTION` relationships. `FRAME` entities become densely connected nodes and queries encountering `FRAME` entities will likely exhibit poor performance. Though the performance of Neo4j will improve in these situations, we may need to revise our model in the interim, perhaps using additional indexing structures. This is particularly necessary given our plan to scale up to tens of thousands of `ACTORS` and tens of millions of `INTERACTIONS`.

6.2 Open Questions

Strongly-typed Relationships

In structures such as our temporal index (see Section 4.2 and Figure 4.2), is it best to connect nodes in the tree using "strongly-typed" relationships (for more details, see *Path Tree* [5]), or as more general relationships qualified by an attribute, i.e., should the temporal (e.g., day, month, year) information be encoded in the relationship type itself, or in properties of the relationship entity. As the strongly-typed solution eliminates the need for property lookups on those relationships it is likely measurably faster than the more general approach. To further illustrate the concept, see the two (equivalent) queries below:

Using strongly-typed relationships:

```
START run = node(some_run_id)
MATCH run-[:HAS_TIMELINE]->tl,
    tl-[:‘2009’]->()-[:‘7’]->()-[:‘1’]->()-[:‘9’]->hour,
    hour-[:TIMELINE_INSTANCE]->frame
RETURN frame
```

Using relationship properties:

```
START run = node(some_run_id)
MATCH run-[:HAS_TIMELINE]->tl,
    tl-[y:NEXT_LEVEL]->()-[m:NEXT_LEVEL]->month,
    month-[d:NEXT_LEVEL]->()-[h:NEXT_LEVEL]->hour,
    hour-[:TIMELINE_INSTANCE]->frame
WHERE y.year=2009 and m.month=7 and d.day=1 and h.hour=9
RETURN frame
```

Removing Unnecessary Entities

The many `RUN_FRAME` and `RUN_INTERACTION` relations may be unnecessary if `INTERACTION` and `FRAME` entities can efficiently indexed and retrieved. The large amount of `RUN_FRAME` and `RUN_INTERACTION` relations negatively impact query performance - they transform `RUN` entities into densely connected nodes.

Dynamic Frames

If `INTERACTIONS` had `START_TIMELINE_INSTANCE` and `END_TIMELINE_INSTANCE` relationships, it may feasible to remove the `FRAME` concept entirely. Dynamic frames could be inferred during a post-processing phase, or interactively at query time, making the model more flexible. However, it is unlikely this would achieve a significant reduction in the number of relationships, as the number of `START`/`STOP` relations would likely be on the same order of magnitude as the number of `FRAMES`. This is due to many of the `INTERACTIONS` flickering on/off from `FRAME` to `FRAME`. Moreover, the `FRAME` concept is a simple abstraction, allowing efficient retrieval of all the `ACTORS`/`INTERACTIONS` that are relevant at a given time. It is an open question as to whether the more general approach could be implemented with comparable efficiency.

7. REFERENCES

[1] GEXF (Graph Exchange XML Format). `http://gexf.net/format/`, 2007.

[2] Neo4j - Collections Library. `https://github.com/neo4j/graph-collections`, 2007.

[3] SocioPatterns - Hypertext 2009 Dynamic Contact Network. `http://www.sociopatterns.org/datasets/hypertext-2009-dynamic-contact-network/`, 2009.

[4] Cypher Graph Query Language. `http://docs.neo4j.org/chunked/stable/cypher-query-lang.html`, 2010.

[5] Neo4j - Data Modeling. `http://docs.neo4j.org/chunked/stable/data-modeling-examples.html`, 2010.

[6] Neo4j Graph Database. `http://www.neo4j.org/`, 2010.

[7] Time-dependent Graphs in Neo4j. `https://github.com/ccattuto/neo4j-dynagraph/wiki/Representing-time-dependent-graphs-in-Neo4j`, 2012.

[8] H. Alani, M. Szomszor, C. Cattuto, W. Van den Broeck, G. Correndo, and A. Barrat. Live social semantics. *The Semantic Web-ISWC 2009*, pages 698–714, 2009.

[9] A.-L. Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435(7039):207–211, 2005.

[10] C. Cattuto, W. Van den Broeck, A. Barrat, V. Colizza, J.-F. Pinton, and A. Vespignani. Dynamics of person-to-person interactions from distributed rfid sensor networks. *PLoS ONE*, 5(7):e11596, 07 2010.

[11] P. Holme and J. Saramäki. Temporal networks. *Physics Reports*, 2012.

[12] L. Isella, J. Stehlé, A. Barrat, C. Cattuto, J.-F. Pinton, and W. V. D. Broeck. What's in a crowd? analysis of face-to-face behavioral networks. *Journal of Theoretical Biology*, 271:166–180, 2011.

[13] M. E. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[14] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.

[15] A. Vazquez. Exact results for the Barabási model of human dynamics. *Physical review letters*, 95(24):248701, 2005.

[16] A. Vázquez, J. G. Oliveira, Z. Dezsö, K.-I. Goh, I. Kondor, and A.-L. Barabási. Modeling bursts and heavy tails in human dynamics. *Physical Review E*, 73(3):036127, 2006.