A study on time models in graph databases for security log analysis

Daniel Hofer
Institute for Application-oriented Knowledge Processing, Johannes Kepler University Linz, Linz, Austria and LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Linz, Austria

Markus Jäger
Pro2Future GmbH, Linz, Austria, and

Aya Khaled Youssef Sayed Mohamed and Josef Küng
Institute for Application-oriented Knowledge Processing, Johannes Kepler University Linz, Linz, Austria and LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Linz, Austria

Received 15 March 2021
Revised 20 May 2021
Accepted 31 May 2021

Abstract

Purpose – For aiding computer security experts in their study, log files are a crucial piece of information. Especially the time domain is very important for us because in most cases, timestamps are the only linking points between events caused by attackers, faulty systems or simple errors and their corresponding entries in log files. With the idea of storing and analyzing this log information in graph databases, we need a suitable model to store and connect timestamps and their events. This paper aims to find and evaluate different approaches how to store timestamps in graph databases and their individual benefits and drawbacks.

Design/methodology/approach – We analyse three different approaches, how timestamp information can be represented and stored in graph databases. For checking the models, we set up four typical questions that are important for log file analysis and tested them for each of the models. During the evaluation, we used the performance and other properties as metrics, how suitable each of the models is for representing the log files' timestamp information. In the last part, we try to improve one promising looking model.

Findings – We come to the conclusion, that the simplest model with the least graph database-specific concepts in use is also the one yielding the simplest and fastest queries.

The research reported in this paper has been mostly supported by the LIT Secure and Correct Systems Lab.

Additionally, this work has partially been supported by the FFG, Contract No. 854184: "Pro^2 Future is funded within the Austrian COMET Programme Competence Centers for Excellent Technologies under the auspices of the Austrian Federal Ministry of Transport, Innovation and Technology, the Austrian Federal Ministry for Digital and Economic Affairs and of the Provinces of Upper Austria and Styria. COMET is managed by the Austrian Research Promotion Agency FFG".

This article is based upon conference paper *On Applying Graph Database Time Models for Security Log Analysis*, hosted on SpringerLink with DOI https://doi.org/10.1007/978–3-030–63924-2_5 (Hofer *et al.*, 2020). We added Section 8 about an attempt on improving the timeline model and extended the descriptions of each query of the evaluation in Section 5.

Research limitations/implications – Limitations to this research are that only one graph database was studied and also improvements to the query engine might change future results.

Originality/value – In the study, we addressed the issue of storing timestamps in graph databases in a meaningful, practical and efficient way. The results can be used as a pattern for similar scenarios and applications.

Keywords Security, Graph database, Logfile analysis, Time model representation

Paper type Research paper

1. Introduction

When analyzing logging information, the time domain is most important, as timestamps are usually the only means for finding corresponding log entries when data from different sources are combined. Therefore, it is crucial to have a data model at hand which allows all sorts of different queries whilst at the same time, the model must not become too complex for performance and maintainability reasons. The overall topic this paper contributes to is to get arguments, whether it is a good idea to use graph databases as combined storage for all systems logs, ranging from one single machine up to a whole enterprise network. Following a bottom-up approach for the overall design of such a data model, the part representing the temporal domain appears to be a good starting point. This paper outlines the aspects our proposed timestamp models are checked against, the data we used for testing and it then analyses three different data model designs for processing time information.

Each model's quality is measured by its expressiveness, the complexity of the queries from a human being's point of view and by the suitability for a graph database using pattern matching for its queries. During the evaluation of these properties, the model must be capable of answering the following questions:

- Q1: For a given event beginning, where is the corresponding ending?
- Q2: Which IP addresses were connected at a given timestamp?
- Q3: Which of the two given log entries occurred earlier?
- Q4: Which events occurred X-time units before/after a given event?

As a testing platform, Neo4J [1] was chosen because it is a native graph database [2] and there exists an open-source community version. Furthermore, a web application called *Neo4J Browser* [3] for developing queries exists which is also open source.

2. Related work

To the best knowledge of the authors, there is no related work dealing with time modelling in graph databases in the specific context of this research. Nevertheless, there are some scientific publications that can be used as anchor points for further literature research in this field.

Theodoulidis and Loucopoulos (Theodoulidis and Loucopoulos, 1991) introduce time modelling concepts in a conceptual schema. Patton (Patton, 2001) tries to model time-varying dependencies between random distributed variables in the context of currencies (exchange rates). Wiese and Omlin (Wiese and Omlin, 2009) give an approach for modelling time with long short-term memory (LSTM) recurrent neural networks in the context of security and safety applications in credit card transactions, fraud detection and machine learning.

A more security-related research field is presented by Schwenk (2014), modelling time for authenticated key exchange protocols. Semertzidis and Pitoura (Semertzidis and Pitoura,

2016) give a first insight into managing historical time queries by using structured data, represented in graph databases.

Recent literature was presented by Maduako and Wachowicz (2019), who model places and events in a network with space-time varying graphs and by Chu *et al.* (2020), who present a novel deep learning method for query task execution time prediction in graph databases.

Using time spacial modelling methods in ontologies, Hobbs and Pan (Hobbs and Pan, 2013) describe an OWL-2 DL ontology of temporal concepts (OWL = web ontology language), which was verified by Grüninger (2011) in 2011.

Further, non-scientific sources (GraphGrid, 2015) and (Bachman, 2013) propose variations of the model introduced in Section 4.3 but do not give any recommendations for which purpose their models should be used or how to query them.

More literature is found on the general use of graph databases for network security monitoring like proposed by Diederichsen *et al.* (2019) or by Tao *et al.* (2018) but without going into detail on the representation of the time domain.

3. Input data

Our testing data originate from a live and personal post-fix e-mail server. A small example of the input data is shown in Listing 1. Due to privacy reasons, the public internet protocol (IP) addresses from the original logging information were replaced by ones from the *Private Address Space* defined in request for comments (RFC) 1918. All other values remained unchanged.

Listing 1 Example of log information used as test data. For privacy reasons, the public IP addresses were replaced by random private ones.

```
1 April 26 05:32:55 smtp postfix/smtps/smtpd[728399]: connect from unknown[10.8.21.42]2 April 26 05:33:01 smtp postfix/smtps/smtpd[728399]: disconnect from unknown [10.8.21.42]3 April 26 05:33:01 smtp postfix/smtps/smtpd[728399]: connect from unknown[10.8.21.42]
```

This type of test data was chosen because it is comparatively easy to read for a human being, all lines share a common format and most importantly, the log file contains events with an explicitly stated beginning and ending. For example, a connection initiation is denoted by *connect from X* and its termination *disconnect from X*. Between these two lines, we consider the connection to be alive.

In the analysis of the graph database models, the following information will be used:

- *Timestamp (Apr 26 05:32:55)*: The main data of interest for our model.
- *Process identifier (728399)*: As multiple remote systems can connect to post-fix simultaneously, multiple instances need to run. As a result, the system wide process identifier or process identifier (PID) is required for matching corresponding lines together.
- *The action executed (connect from/disconnect from)*: Denoting the beginning and ending of each event or in the following connection.
- *The remote internet protocol (IP) address (10.8.21.42)*: Identifying the remote client.

By default, the timestamps produced by post-fix only have a granularity down to seconds. As a result, multiple lines in the log file, each representing one event, can share the same timestamp, rendering their order indistinguishable if we are only relying on the timestamp information. To compensate this problem, we assume that the relative order of one line compared to its neighbors is known during processing, and therefore, must be encoded in the data model.

4. Our data models

We propose three different data models for storing timestamp information in graph databases. The first one is basically a one-to-one translation originating from relational databases. These would store log lines in rows and the timestamp would be one column. In the graph database world, we store the time information as properties (the equivalent of the column) in the nodes (the equivalent of the rows) representing events. In our case, these are nodes to represent *connection* and *disconnection*. Our second model still keeps the timestamp information inside one dedicated node for each entry, but the order of nodes is represented by directed edges between the timestamp nodes, basically forming one long timeline. For the third model, we exploit the hierarchical structure of a timestamp, basically building a tree like structure.

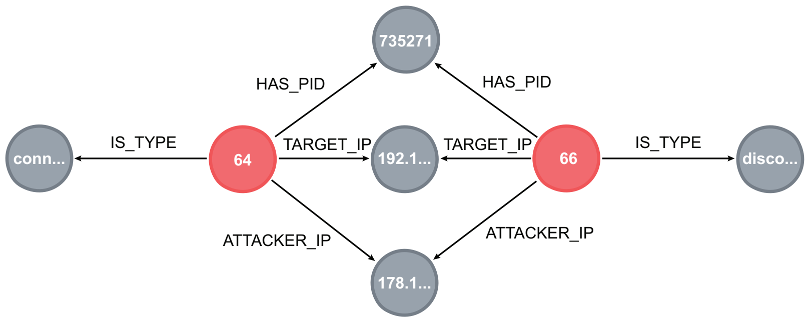
Most parts of the models are common for all three versions. For this reason, they are described here once for all using generic names. In the remainder of this work, to make nodes of one model distinguishable from other models, the naming convention *<model prefix>_<node type name>* is used. The *model prefix* consists of two letters loosely derived from the model's name. In the case of the *property-based model* the prefix is *Pt*, for the *timeline model*, *Tl* is used and the *hierarchical model* is marked with *Hr*.

The core of all models are the *Event* nodes. Each node of this type corresponds to one *connect* or *disconnect* line in the processed log file. As the remaining properties like involved IP addresses in each log line are reused, this information is modelled in dedicated nodes (except for the timestamps of course). Edges are usually directed from the *Event* node to the referenced/shared nodes. Remote IP addresses are represented by *Ip* nodes, connected to the event node by *ATTACKER_IP* edges. Whilst the local IP also uses this type of node, it is connected using *TARGET_IP* edges. The type of the event (connecting or disconnecting) is expressed by *IS_TYPE* edges from said event node to *Type* nodes. To aggregate events by *PID*, *Process* nodes in conjunction with *HAS_PID* edges are used. Because a productive server environment might be up for a long time, the process ID is bound to be reused at some point in time. Whilst this has to be taken into account in the real world, the currently observed time frame is too short so that a PID rollover can be neglected for the observations in this work (cnicutar, 2020; Chazelas, 2020).

An example for the common part of the data model based on the *property-based model* can be found in Figure 1.

4.1 Property-based model

For this very basic model, the complete timestamp information is solely one property contained in the *Pt_Event* node. Neo4J offers support for this by providing several dedicated



Note: The red nodes (64 and 66) represent a log line for the property-based model in this example

Figure 1.
Part of the graph structure common for all three versions of the data model

datatypes (Neo4j, I, 2020a). Apart from the timestamp itself, additional information is required due to the coarse granularity of whole seconds for the timestamp. As a result, the order for multiple events with the same timestamp is not distinguishable anymore. To work around the problem, the property *order* is introduced. It contains an increasing number starting at 1 and denotes the position in the logfile for a group of events sharing the same timestamp. This property can then be used by the query to sort the events in such a group. An example for an event node of the *property-based model* can be seen in Figure 2.

4.2 Timeline model

In the *timeline model*, every log line from the input files receives a dedicated timestamp node. All these nodes are connected by *NEXT* edges pointing towards younger timestamps. Event nodes are then attached using *HAPPENED_AT* edges to their dedicated timeline nodes (Figure 3). As for the *property-based model*, again problems concerning event order on equal timestamps arise. For this model, our solution is to allow multiple timestamp-nodes per actual timestamp. The order is maintained by the directed *NEXT* edges.

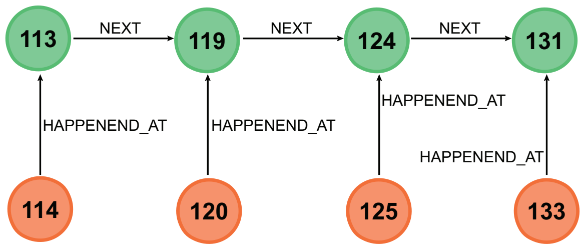
4.3 Hierarchical model

A timestamp itself is some kind of hierarchical composition. Every year has a fixed amount of months which have a certain number of days, etc. The *hierarchical model* tries to exploit this relationship by creating dedicated nodes for each level from years down to the desired granularity. Every time unit is connected to the more granular one with *FINER* edges. For lateral movement, *NEXT* edges are used for connecting nodes in ascending order, but only if they share the same parent node. Afterwards, for the sake of simplicity, we will refer to a part of the resulting structure as a *time tree*, although the nodes on the year level do not share an existing parent node, and

Pt_Event **<id>:** 64 **order:** 1 **timestamp:** 2020-04-27T05:57:27.000Z

Note: The timestamp information is encoded as property inside the node using a dedicated datatype

Figure 2.
Example for an event node of the *property-based model*



Note: Nodes in the upper row contain timestamp information, the ones from the lower row represent events

Figure 3.
Example of the *timeline model*

therefore, only one year is a formally sound tree. An example for a part of the resulting structure can be seen in Figure 4.

The *Hr_Event* nodes are then attached to the nodes representing *seconds* using *HAPPENED_AT* edges. For solving the problem of equal timestamps, event nodes are connected with *NEXT_EVENT* edges, if they belong to the same process. Edges are again directed from older event nodes to younger ones. The name is chosen differently to *NEXT* to make the difference explicit and avoid confusion during queries. In Figure 5, a whole example is shown with several event nodes belonging to the same process and partly sharing the same *Hr_Second* node.

As outlined in *related work* at Section 2, in non-scientific literature (Bachman, 2013; GraphGrid, 2015) the same and a very similar model is described. Especially (Bachman, 2013) contains not only the same structure for the timestamp part as proposed here but also enhances it by adding *FIRST* and *LAST* edges directing from a time unit to the next finer one.

5. Evaluation

In this part, we will propose specialized queries for the questions formulated in Section 1 for each of the previously defined data models.

In an attempt of increasing the query's readability and genericity, we tried to split each of them in two parts. The first part is only responsible for fetching a specific event node by different means. Our proposed examples here are simply using the ID of an already known node or specific values. In any case, this part of the query delivers the event nodes or other data a user might wish to operate on in a generic way to the remainder of the query so that the implementation details are easily exchanged according to a use-case. The second part does the actual work answering the user's question. This part is specific for the data model and the contained data, but should not require changes when working for different applications.

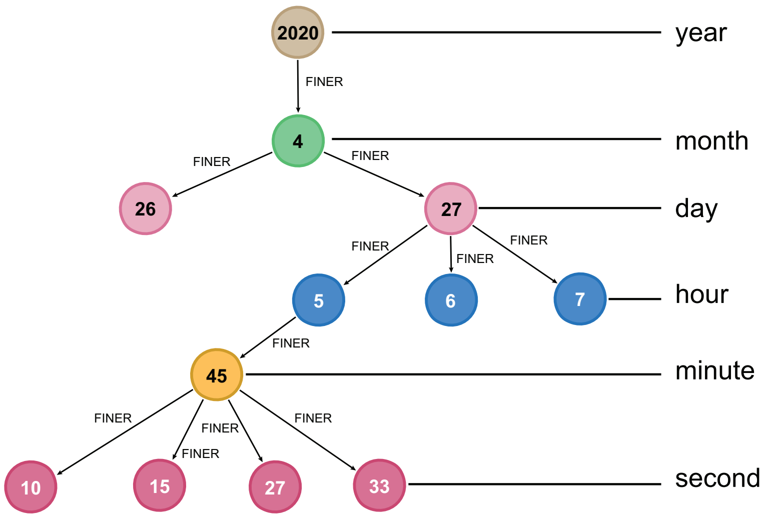
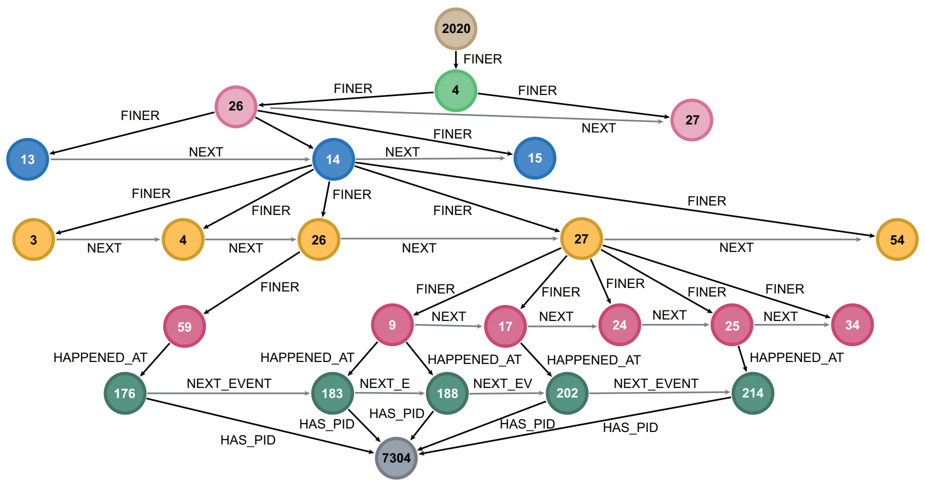


Figure 4.
Example for a part of a basic timestamp tree

Note: The top node represents the year and the FINER edges lead to the nodes with smaller time granularity



Notes: The nodes in the penultimate line represent log lines whilst the single node at the bottom stands for a process. The remaining structure expresses timestamps as described in Figure 4

Figure 5.
Complete example of the hierarchical model with several event nodes

In some cases, it might be possible to write a shorter expression for a query. Again, with the intention to create more readable versions of the queries, the proposed ones have been chosen.

Under the assumption, that not only connect and disconnect events are present but also other types, it is always necessary to filter for the desired event node type. However, it is still assumed a single process can only handle a single remote connection at a time.

5.1 Question 1: for a given event beginning, where is the corresponding ending?

For the first question, the beginning of an event is known, which is in the actual test data an incoming connection and the user wants to know when this event ends, which means for this example, when the remote machine disconnects. The *connect* event node is found by a hard coded ID, which simulates that a specific connection is currently under investigation and the matching disconnect has to be found.

5.1.1 Property-based model. In Listing 2, the disconnect log entry is found by the shared process ID between connection and disconnection log entry. As the process cannot handle two connections simultaneously, filtering for nodes with the same process ID as shown in Line 5 is enough, if only the node is returned which is from a time perspective point of view the closest one.

Listing 2 Finding the *disconnect* event node for a given *connects* in the *property-based model.*

```
1 MATCH (connect:Pt_Event) WHERE id(connect) = 17 WITH connect
2
3 MATCH
4 (connect:Pt_Event), (process:Pt_Process), (disconnect:Pt_Event),
5 (connect)-[:HAS_PID]->(process)<-[:HAS_PID]-(disconnect),
6 (disconnect)-[:IS_TYPE]->(:Pt_Type{name: "disconnect"})
7 WHERE connect.timestamp <= disconnect.timestamp
```

```
8
9 RETURN disconnect
10 ORDER BY disconnect.timestamp ASC, disconnect.order ASC LIMIT 1
```

5.1.2 Timeline model. The query for the timeline model in Listing 3 is slightly more complicated. It works by exploiting the single timeline bringing all time nodes in one global order. As the disconnect event must have occurred later, the search can make use of the directed edges (see Line 8). The shortest paths between the given start node and all satisfying *disconnect* nodes are queried there. By selecting the minimal one, the connection ending is found.

Listing 3 Finding the *disconnect* event node for a given *connects* in the *timeline model.*

```
1 MATCH (connect:Tl_Event) WHERE id(connect) = 172 WITH connect
2
3 MATCH
4 (connect)-[:HAPPENED_AT]->(connectTimestamp),
5 (disconnect)-[:HAPPENED_AT]->(disconnectTimestamp),
6 (disconnect:Tl_Event)-[:IS_TYPE]->(:Tl_Type{name: "disconnect"}),
7 (connect)-[:HAS_PID]->(t:Tl_Process)<-[:HAS_PID]-(disconnect),
8 p = shortestPath((connectTimestamp)-[:NEXT*1.c]->
     (disconnectTimestamp))
9 WITH disconnect, length(p) AS len ORDER BY len ASC LIMIT 1
10
11 RETURN disconnect
```

5.1.3 Hierarchical model. For finding the disconnect log entry in the hierarchical model, there are basically two different approaches possible.

The first is denoted in Listing 4. This query works by finding the shortest path in the *time tree* but unlike the previous query for the *timeline model*, it is not possible to indicate the direction, as the search must also go up the hierarchy. To ensure that only events after the *connect* are returned, Line 8 is used which at the same time restricts to the right process.

Listing 4 Finding the *disconnect* event node for a given *connect* in the *hierarchical model* using the time information.

```
1 MATCH (connect:Hr_Event) WHERE id(connect) = 121 WITH connect
2
3 MATCH
4 (disconnect:Hr_Event), (ip:Hr_Ip), (connect:Hr_Event),
5 (disconnect)-[:ATTACKER_IP]->(ip)<-[:ATTACKER_IP]-(connect),
6 (connect)-[:HAPPENED_AT]->(con_sec:Hr_Second),
7 (disconnect)-[:HAPPENED_AT]->(dis_sec:Hr_Second),
8 (connect)-[:NEXT_EVENT*]->(disconnect),
9 (disconnect)-[:IS_TYPE]->(connectType:Hr_Type {name:
     "disconnect"}),
10 p = shortestPath((con_sec:Hr_Second)-[:FINER|NEXT*0..]-
     (dis_sec:Hr_Second))
11
12 RETURN disconnect ORDER BY length(p) ASC LIMIT 1.
```

The part of the hierarchical model indicating the precedence of a event node over the other (the *NEXT_EVENT* edges) is basically a special case of the *timeline model*. As a result, it is possible to adapt the query from Listing 3 and create a slightly less complicated query shown in Listing 5. The main difference is that the detour over the *time tree* is not taken and instead the information encoded by the *NEXT_EVENT* edges is sufficient. Nevertheless, for the performance test later in Section 6.1, the first query from Listing 4 is used.

Listing 5 Finding the *disconnect* event node for a given *connect* in the *hierarchical model* using the log entry precedence information.

```
MATCH (connect:Hr_Event) WHERE id(connect) = 121 WITH connect,

MATCH
disconnect:Hr_Event), (connect:Hr_Event), (ip:Hr_Ip),
(disconnect)-[:ATTACKER_IP]->(ip)<-[:ATTACKER_IP]-(connect),
(disconnect)-[:IS_TYPE]->(connectType:Hr_Type {name:
    "disconnect"}),
p = shortestPath((connect)-[:NEXT_EVENT*..]->(disconnect))
RETURN disconnect ORDER BY length(p) ASC LIMIT 1
```

5.2 Question 2: which IP addresses were connected at a given timestamp?

For this question, it is required that the event type under investigation lives over a measurable timespan, which is the case for a connection bound by the *connect* and *disconnect* log entries and their associated event nodes. As input data, a point in time is given and as output, a list of IP addresses is expected which were connected at the specified point in time. In the following examples, the timestamp *2020–04-26 06:30:17 CEST* is used. As *datetime* objects of Neo4J are timezone aware, a conversion to UTC might be required and is implicitly done if required (Neo4j, 2020a). It is also notable that the selected timestamp is not represented directly by any node because nothing happened at this timestamp in the input log file. As a result, it is not possible to directly use a node representing the desired timestamp for the subsequent queries.

5.2.1 Property-based model. The *property-based model* can work with the provided timestamp directly, as the time information is, in any case, not stored in dedicated nodes. In Listing 6, the aim of the query, apart from the usual restrictions, is to create a temporary *datetime* object and compare all nodes in the database against it.

Listing 6 Finding the currently connected IP addresses for a specified point in time for the *property-based model*.

```
WITH datetime("2020-04-26T06:30:17.000[Europe/Vienna]") AS
    timestamp

MATCH
(connect:Pt_Event), (process:Pt_Process), (disconnect:Pt_Event),
(connect)-[:HAS_PID]->(process)<-[:HAS_PID]-(disconnect),
(connect)-[:IS_TYPE]->(:Pt_Type{name: "connect"}),
(disconnect)-[:IS_TYPE]->(:Pt_Type{name: "disconnect"})
WHERE
connect.timestamp <= timestamp AND
disconnect.timestamp >= timestamp AND
(
  connect.timestamp < disconnect.timestamp
  OR
  connect.timestamp = disconnect.timestamp AND
  connect.order < disconnect.order
)
WITH disconnect
MATCH (disconnect)-[:ATTACKER_IP]->(ip)
```

```

RETURN DISTINCT ip.ip AS connected_ips
```

5.2.2 Timeline model. Finding the connected IP addresses at a specific point in time in Listing 7 requires several steps. It works by first filtering all *connect* events before the given timestamp. For all resulting nodes, possible disconnection nodes are found and the shortest paths between them are calculated (multiple per connection node). In the next step, for each *connected* node, only the shortest path is retained. Based on this result, in the third step, all the *disconnect* nodes which have occurred before the desired timestamp are filtered out. The resulting nodes now have a connection before and disconnection after the desired timestamp. Now, the remaining task is fetching the remote IP addresses.

Listing 7 Finding the currently connected IP addresses for a specified point in time for the *timeline model*.

```
WITH datetime("2020-04-26T06:30:17.000[Europe/Vienna]") AS
    timestamp

MATCH
(connect:Tl_Event)-[:HAPPENED_AT]->
    (connectTimestamp:Tl_Timestamp),
(connect)-[:IS_TYPE]->(:Tl_Type{name: "connect"}),
(disconnect:Tl_Event)-[:HAPPENED_AT]->
    (disconnectTimestamp:Tl_Timestamp),
(disconnect)-[:IS_TYPE]->(:Tl_Type{name: "disconnect"}),
(connect)-[:HAS_PID]->(:Tl_Process)<-[:HAS_PID]-(disconnect),
p = shortestPath((connectTimestamp)-[:NEXT*1..]->
    (disconnectTimestamp)),
WHERE
connectTimestamp.timestamp <= timestamp

WITH timestamp, connect, p ORDER BY length(p) ASC
WITH timestamp, connect, collect(p) AS paths
WITH timestamp, connect, paths[0] AS p
UNWIND nodes(p) AS disconnectTimestamp,
WITH timestamp, connect, disconnectTimestamp

MATCH
(connect)-[:HAS_PID]->(:Tl_Process)<-[:HAS_PID]-(disconnect),
(disconnect)-[:HAPPENED_AT]->(disconnectTimestamp),
(disconnect)-[:ATTACKER_IP]->(ip:Tl_Ip)
WHERE disconnectTimestamp.timestamp >= timestamp

RETURN ip.ip AS connected_ips
```

5.2.3 Hierarchical model. For finding the currently connected remote machines at a point in time in the *hierarchical model*, the approach from the *timeline model* is slightly adapted in Listing 8. Because this model does not support different timezones by default, the reference timestamp has to be adapted accordingly compared to the other models, which means adding 2 h to convert UTC to CEST. Apart from this, the main difference is the way the time information is processed, as the values for the *datetime* objects have to be extracted from the *time tree* first.

Listing 8 Finding the currently connected IP addresses for a specified point in time for the *hierarchical model*.

```
WITH datetime("2020-04-26T06:30:17.000Z") AS timestamp

MATCH
(year:Hr_Year)-[:FINER]->(month:Hr_Month)
-[:FINER]->(day:Hr_Day)
-[:FINER]->(hour:Hr_Hour)
-[:FINER]->(minute:Hr_Minute),
-[:FINER]->(second:Hr_Second),
(connect:Hr_Event)-[:HAPPENED_AT]->(second),
(connect)-[:IS_TYPE]->(:Hr_Type{name: "connect"}),
(disconnect:Hr_Event)-[:IS_TYPE]->(:Hr_Type{name:
    "disconnect"}),
p = shortestPath((connect)-[:NEXT_EVENT*1..]->(disconnect)),
WHERE
datetime({year: year.value, month: month.value, day:
    day.value, hour: hour.value, minute: minute.value, second:
    second.value}) <= timestamp
WITH timestamp, connect, disconnect, p

WITH timestamp, connect, disconnect, p ORDER BY length(p) ASC
WITH timestamp, connect, collect(p) AS paths
WITH timestamp, connect, paths[0] AS p
UNWIND nodes(p) AS disconnect
WITH timestamp, connect, disconnect

MATCH
(year:Hr_Year)-[:FINER]->(month:Hr_Month)
-[:FINER]->(day:Hr_Day)
-[:FINER]->(hour:Hr_Hour)
-[:FINER]->(minute:Hr_Minute)
-[:FINER]->(second:Hr_Second),
(connect)-[:HAS_PID]->(:Hr_Process)<-[:HAS_PID]-(disconnect),
(disconnect)-[:HAPPENED_AT]->(second),
(disconnect)-[:ATTACKER_IP]->(ip:Hr_Ip)
WHERE
datetime ({year: year.value, month: month.value, day: day.value,
    hour: hour.value, minute: minute.value, second:
    second.value}) >= timestamp

RETURN ip.ip AS connected_ips
```

5.3 Question 3: which of the two given log entries occurred earlier?

For answering this question, two *Event* nodes are given by ID and the query answers, which of the provided log entries occurred earlier. This time, the query solely uses the timestamp and ignores the precedence information. The query takes the first given node A as a reference and answers whether the second one B is less, equal or greater.

5.3.1 Property-based model. Finding the query for the actual question is as easy as Listing 9 which consists of selecting two nodes and then comparing their timestamp values.

Listing 9 Answering which event occurred before/after the other one in the *property-based model.*

```
MATCH (a), (b) WHERE id(a) = 168 AND id(b) = 48,

RETURN
a.timestamp > b.timestamp AS less,
a.timestamp = b.timestamp AS equal,
a.timestamp < b.timestamp AS greater
```

5.3.2 Timeline model. The query for the *timeline model* basically works like the previous one and is shown in Listing 10. Mainly, the time comparison is based on the existence of a directed *NEXT* path between the two nodes in question. For the case of value equality, an appropriate check is done. This leads to the unique property that the *equals* answer can be true in addition to *less* and *greater*.

Listing 10 Answering which event occurred before/after the other one in the *timeline model.*

```
MATCH (a), (b) WHERE id(a) = 51 AND id(b) = 140 WITH a, b,

MATCH
(a)-[:HAPPENED_AT]->(aTimestamp:Tl_Timestamp),
(b)-[:HAPPENED_AT]->(bTimestamp:Tl_Timestamp),

RETURN
EXISTS((aTimestamp)<-[:NEXT*]-(bTimestamp)) AS less,
aTimestamp.timestamp = bTimestamp.timestamp AS equals,
EXISTS((aTimestamp)-[:NEXT*]->(bTimestamp)) AS greater
```

5.3.3 Hierarchical model. The answer for the precedence of event nodes in the *hierarchical model* is determined by the query in Listing 11.

Here, also a form of path existence check is used, but this time with the *optional match.* The usage of *optional* allows the query to set the path variables to *null* if no appropriate path is found. Like for the *timeline model*, the directions of *NEXT* edges are used, but to overcome the presence of the hierarchy of the *time tree*, the empty nodes, which are denoted by *()*, are required. Also, it has to be differentiated between the case where the shortest path is built using *NEXT* edges and a special case where only a common parent node is found.

Listing 11 Answering which event occurred before/after the other one in the *hierarchical model.*

```
MATCH (a)-[:HAPPENED_AT]->(timestampA) WHERE id(a) = 119 WITH *
MATCH (b)-[:HAPPENED_AT]->(timestampB) WHERE id(b) = 136 WITH *

OPTIONAL MATCH,
p = ((timestampA)<-[:FINER*0..]-()<-[:NEXT*]-()-[:FINER*0..]->
    (timestampB))
WITH *

OPTIONAL MATCH
q = ((timestampA)<-[:FINER*0..]-()-[:NEXT*]->()-[:FINER*0..]->
    (timestampB))
WITH *

OPTIONAL MATCH
```

```
r = ((timestampA)<-[:FINER*0..]-(commonA)<-[:FINER]-()-[:FINER]->
     (commonB)-[:FINER*0..]->(timestampB))

RETURN
p IS NOT NULL OR commonA.value > commonB.value AS less,
timestampA = timestampB AS equal,
q IS NOT NULL OR commonA.value < commonB.value AS greater.
```

5.4 Question 4: which events occurred X time units before/after a given event?

When trying to find the root cause of an event, it may be of interest, which events occurred in a timeframe before or after a specific event. Accordingly, the ID and timeframe of an event are given as input for this question (in this example, 1 h ago). The expected results for the queries are all event nodes enclosed by the calculated timestamp on one hand and the previously given event on the other hand.

5.4.1 Property-based model. For answering the question in the property-based model, Listing 12 starts by querying an event node by ID and extracting the timestamp. With this information, the second timestamp is calculated and then used for filtering all events in the database.

Listing 12 Finding all events in a certain time frame in the *property-based model.*

```
MATCH (event:Pt_Event) WHERE id(event) = 1267
WITH event.timestamp AS timestamp

WITH
datetime ({year: timestamp.year, month: timestamp.month, day:
     timestamp.day, hour: timestamp.hour-1, minute: timestamp.
     minute, second: timestamp.second}) AS lowerTimestamp,
timestamp AS upperTimestamp

MATCH (events:Pt_Event)
WHERE
lowerTimestamp <= events.timestamp AND
events.timestamp <= upperTimestamp

RETURN events
```

5.4.2 Timeline model. The *timeline model* version is shown in Listing 13. As it is unknown whether a *timestamp* node representing the second boundary exists, the timeline can only be exploited for one boundary and for the second, roughly the same approach as for the *property-based model* is applied.

Listing 13 Finding all events in a certain time frame in the *timeline model.*

```
MATCH (event:Tl_Event)-[:HAPPENED_AT]->(timestamp:Tl_Timestamp)
WHERE id(event) = 1179
WITH timestamp, timestamp.timestamp AS timeVar

MATCH
(earlierTimestamp:Tl_Timestamp)-[:NEXT*0..]->(timestamp),
(event:Tl_Event)-[:HAPPENED_AT]->(earlierTimestamp),
WHERE
earlierTimestamp.timestamp >= datetime ({year:timeVar.year, month:
     timeVar.month, day:timeVar.day, hour:timeVar.hour-1, minute:
     timeVar.minute, second:timeVar.second})
```

```

RETURN event
```

5.4.3 Hierarchical model. In the hierarchical model, we were not able to exploit the actual data structure at all. Instead, the solution in Listing 14 consists of an approach in which the data is read and converted from the hierarchical model and then processed like in the *property-based model*.

Listing 14 Finding all events in a certain time frame in the *hierarchical model*.

```
MATCH (event:Hr_Event)-[:HAPPENED_AT]->
    (second:Hr_Second)<-[:FINER]-
    (minute:Hr_Minute)<-[:FINER]-
    (hour:Hr_Hour)<-[:FINER]-
    (day:Hr_Day)<-[:FINER]-
    (month:Hr_Month)<-[:FINER]-
    (year:Hr_Year)
WHERE id(event) = 2150

WITH
datetime ({year: year.value, month: month.value, day: day.
    value, hour: hour.value, minute: minute.value, second:
    second.value}) AS upperTimestamp,
datetime ({year: year.value, month: month.value, day: day.
    value, hour: hour.value-1, minute: minute.value, second:
    second.value}) AS lowerTimestamp,

MATCH
    (event:Hr_Event)-[:HAPPENED_AT]->
    (second:Hr_Second)<-[:FINER]-
    (minute:Hr_Minute)<-[:FINER]-
    (hour:Hr_Hour)<-[:FINER]-
    (day:Hr_Day)<-[:FINER]-
    (month:Hr_Month)<-[:FINER]-
    (year:Hr_Year)
WITH lowerTimestamp, upperTimestamp, event,
datetime ({year: year.value, month: month.value, day: day.
    value, hour: hour.value, minute: minute.value, second:
    second.value}) AS timestamp
WHERE lowerTimestamp <= timestamp AND timestamp <=
    upperTimestamp

RETURN event
```

5.5 Summary of the evaluation

During this evaluation, we saw the necessary steps to get an answer for our queries stated in the introduction in Section 1. We also saw, that there is a tendency for the easier models to require less complicated queries, which means the *property-based model* needs smaller queries whilst the *hierarchical model* the more convoluted ones. Furthermore, the long *NEXT* chain considered to be an advantage of the *timeline model* is not always usable and, in some cases, an approach similar to the *property-based model* is required.

6. Comparison of the models

In the introduction, we set up some criteria we want the three different models to compare against. The following only applies to the timestamp part of the models, whereas the remaining information like *EventType* or *Process* is not of interest here.

Table 1 contains three dimensions for each model. *Expressiveness* covers, what kind of semantics can be encoded inside the model. More accurately, this draws the line between implicit knowledge located outside the model and knowledge explicitly stored and displayed by the model. An example: a user of the *property-based model* must know, how timestamps in the nodes can be accessed and how they have to be sorted. In contrast, the *timeline model* already covers this knowledge by explicitly providing directed edges between the nodes. With *complexity*, mainly the effort required by the user for writing the query is described. However, as we can conclude from (Neo4j, I, 2020b), the *with* keyword divides the query into several logical parts, limiting the possibilities for the query optimizer. Therefore, a query containing a lot of *with* statements may also be costly to execute, which also flows to some extent into the *complexity* measure. The row for *suitability* takes into account, that graph databases work by analyzing the structure of stored data and comments and whether such data is present in a usable form.

6.1 Performance measurements

For our performance measures, we used the same data set for all models and selected one fixed event per query, covering some corner cases like a disconnect with a reconnect in the same second. Before the test, we find them on all three different implementations and retrieve their IDs. We then use this IDs in the queries proposed in Section 5. The measurements consist of executing the query 1,000 times as warm up allowing caches and optimizations to adapt and the execution times to converge, already with time measurement in place. Immediately following, we execute the queries a further 10,000 times for the actual measurement. Finally, the arithmetic mean is used on the measurement batch, yielding the numbers shown in Table 3 and for better comparability visualized as bar

	Property-based model	Timeline model	Hierarchy model
Expressiveness	Low, simple property with dedicated datatype (only storage, no semantic information), supports time zones, the property is not unique, needs an additional property to resolve timestamp equality, the user must know how to interpret stored data	Medium, simple property with dedicated datatype in dedicated node, supports time zones, the node can be unique, semantic information: absolute temporal order given by directed edges	High, tree like structure with shared nodes, each node is unique, allows indication of timestamp precision, fine grained node fetching possible, no time zone support, the relative temporal order of nodes of the same level and sub tree by directed edges, requires directed edge to resolve timestamp equality
Complexity	Low, comparable complex as the equivalent query in a relational database	Medium, the problem described in Section 7.1 forces usage of two different approaches for node access and matching	High requires complex queries due to different levels when searching specific nodes
Suitability	Does not exploit graph database specific functionality	Basic usage of nodes and directed edges by constructing a global timeline	Extensive usage of nodes and edges, most information expressed by these elements

Table 1. Comparison of the different data models

charts in Figure 7. These numbers are measured using the full-sized data set. Respectively, Table 2 and Figure 6 show the results of the measurements for a data set which is approximately half of the full set.

The abbreviations used in the tables are the same as described in Section 5 and *Rl* denotes the relational database included for reference. All numbers are given in milliseconds.

6.2 Description of the test setup

As data set, we used logging output from a post-fix mailserver as described in Section 3. In total, we had 3,394 events, splitting up to 1,697 connect- and 1,697 disconnect-events. To determine the growth of required computing time, we did two runs of the performance measurements, one with 1,582 total events and another one with all 3,394 events. We did not use a split at 1,697 events because we wanted to use a point in time for the split at which no connection was active.

The test hardware consisted of an *HP Elitebook 850 G6* with an *Intel SSDPEKNW512G8H* SSD and 16 GB of main memory, running *Arch Linux* with the kernel version *5.11.13-arch1-1*. The used database was *Neo4J Community 4.2.5*. Tests were performed by a *Spring Boot 2.3.0. RELEASE* project, using the *Neo4J JDBC 4.0.0* driver. Furthermore, the relational database used for reference was *MariaDB 10.5.9–1* and the driver was the one delivered with *Spring Boot Data 2.3.0. RELEASE*.

Table 2. Absolute values of Figure 6 (1,582 events in the database)

	Rl	Pt	Tl	Hr
Q1	0.17	0.70	0.67	0.54
Q2	1.36	10.55	3.65	67.52
Q3	0.10	0.59	0.86	1.66
Q4	0.80	4.41	2.84	41.46

Table 3. Absolute values of Figure 7 (3,394 events in the database)

	Rl	Pt	Tl	Hr
Q1	0.20	0.77	0.70	0.63
Q2	3.68	21.18	5.61	138.85
Q3	0.12	0.62	0.89	1.26
Q4	1.57	7.94	2.90	76.74

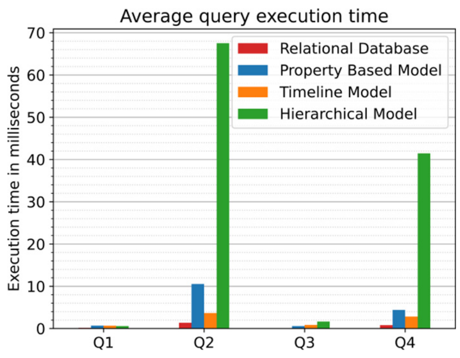


Figure 6. Average execution time for each query in milliseconds with 1,582 events in the database

7. Findings

The following section contains some remarkable findings which arouse during the investigation of the three timestamp representations.

7.1 How to reference arbitrary timestamps

A major problem during the work with the three models was, how an arbitrary timestamp can be accessed. Whilst this is not an issue for the *property-based model* because every representation of a timestamp is only an instance of *datetime* with no further dedicated objects, a fully-fledged representation of a custom timestamp not stored in the database can be created instantly within the query without changing the database.

This is not the case for the *timeline* and *hierarchical model*. These two require at least one dedicated node per represented timestamp. As a result, we are not able to use any arbitrary timestamp in a query because this timestamp may not exist in the database and would have to be created. Questions suffering from this problem are described in Sections 5.2 and 5.4.

7.2 Encoding the order of entries is mandatory

It is not sufficient to only rely on the timestamps saved to each log entry. For one log source, this problem might be omitted by adding precision to the timestamp, but in an environment producing large amounts of log lines, also this approach is not guaranteed to work. Latest when log entries from multiple machines need to be merged, precise timestamps require a sufficient synchronization between all involved clocks. According to (Aichhorn *et al.*, 2017), clock synchronization in a local area network (LAN) using Linux in its default implementation came down to 8 to 10 μs and with their proposed implementation down to 2 μs, which might still not be accurate enough, as a thousand events in one second for a large enterprise are a very realistic number.

During our experiments without a sufficient encoding of log entry order, matching errors did occur. For example, the question from Section 5.1 required precise information which events happened after itself. Otherwise, a disconnect before the following reconnect could mistakenly be recognized as the nearest event *after* the reconnect and wrongly be paired together.

7.3 Complexity of queries

Although this is not backed by metrics, the overall complexity of queries per question seems to differ, whereas the *property-based model* being the model with the shortest queries, the *timeline model* being an intermediate one and the *hierarchical model* yields the most complex queries.

7.4 Expressiveness of the models

The idea behind modelling timestamps in graph databases was in exploiting graph database specific characteristics for easier handling of the stored data. Due to the problem described in Section 7.1, with our current capability, we are in some cases (especially 5.2 and 5.4) not able to make full use of graph database specific query techniques. Instead, the query must force data into a representation which is already well used in relational databases.

Furthermore, one idea behind the *hierarchical model* was being able to represent different levels of precision by attaching event nodes to the according level. An approach which is also proposed in (Bachman, 2013). Attaching and matching event nodes on different levels might increase the queries complexity even more but assumption requires further investigation.

8. Attempt on improving the timeline model

An assumption we made about the *timeline model* is, that graph databases can use their unique modelling capabilities for improved performance during queries on this model. In Figure 7 we saw that the *property-based model* is faster on query *Q3* and *Q4*, the *timeline model* performs better on *Q2* and the result for *Q1* is approximately the same for both models.

One finding, previously proposed in Section 7.1, was, that the *timeline model* suffers from the fact that out of the box, it is not possible to use arbitrary timestamps during queries. This is especially the case for the queries *Q2* and *Q4* which are the two in which the *timeline model* in its current version performs worst. Among the four queries, *Q2* (IPs connected at a timestamp) and *G4* (events X time units ago) are also the two working with time spans. Our idea is when the *timeline model* variants of the queries are able to exploit the *NEXT* edges instead of filtering using the *Tl_Timestamp.timestamp* properties, the performance of the *timeline model* might improve for the performance of these two queries.

The necessary steps are the following: At first we need to find a way to convert an arbitrary timestamp into entities which are already present in the *timeline model*. These entities, in our case nodes with the label *Tl_Timestamp*, can then be used instead of filtering all nodes based on the properties. Of course, the queries need to be adapted accordingly so that finally, we can again do performance measurements.

8.1 Converting arbitrary timestamps to entities of the timeline model

Because timestamps in the *timeline model* are represented by dedicated nodes, the conversion needs to find the best fitting node for the desired timestamp. We have the following three possible cases:

- *Direct hit*: The timestamp we need to convert is already present in the database and represented by a Tl_Timestamp node.
- *Unique neighbours*: The timestamp is not already stored in the database, but there are unique nodes representing a preceding and a successive timestamp. We are safe to conclude that nothing happened exactly on our timestamp, and therefore, fetch the neighbouring nodes with the minimal difference.
- *Ambiguous nodes*: In this case, there is more than one direct hit in the database. This case occurs, when the accuracy of the stored or queried timestamp is not precise enough and the represented events, therefore, appear to have happened simultaneously. As a result, more than one pair of neighbours is valid for selection.

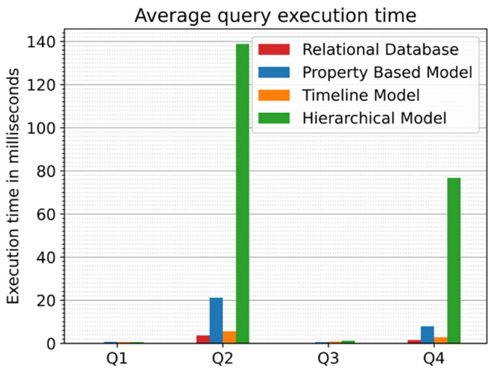


Figure 7.
Average execution time for each query in milliseconds with 3,394 events in the database

This conversion can be achieved with a dedicated part in the query as is shown in Listing 15.

Listing 15 Cypher queries for converting an arbitrary timestamp into the closest representation contained in the database.

```
1 WITH
2 datetime("2020-04-26T06:30:18.000[Europe/Vienna]") AS timestamp
3
4 MATCH
5 (a:Tl_Timestamp)-[:NEXT*0..1]->(b:Tl_Timestamp)
6 WHERE 7 a.timestamp < timestamp AND timestamp < b.timestamp
8 OR
9 a.timestamp = b.timestamp AND a.timestamp = timestamp
10
11 RETURN a, b
```

Line 5 tells the database, that we require two *Tl_Timestamp* nodes called *a* and *b* which are at most one *NEXT* edge away from each other but might also be the same. Their relationship to the *timestamp* variable introduced in Line 2 is then established in the *WHERE* part in Lines 7–9. In Line 7, the case is covered in which the nodes *a* and *b* are not the same and we do not have a direct hit in the database. This part delivers the next smaller and larger nodes for our timestamp in the case of *unique neighbors*. Line 9 is true for the case that the timestamp to convert is already contained in the database (*direct hit*) or when multiple nodes with the same timestamp value exist (*ambiguous nodes*). In the latter case, all neighbouring pairs with the same timestamp are returned. For example, if the nodes *A*, *B* and *C* represent the same timestamp and are connected in this order by *NEXT* edges, the result would consist of *(A, B)* and *(B, C)*.

For the sake of completeness, we have to mention that the query in Listing 15 has not been tested regarding the ends of the *NEXT* chain. Furthermore, in some cases, this conversion from timestamp to two *Tl_Timestamp* nodes is not fully equivalent, as it may introduce new results. An analogy of the problem is the choice of $\leq$ instead of $<$ in a Boolean expression.

8.2 Adapting Q2 to make use of the next edge chain

We can now use the query from the previous step and combine it with a part of the query proposed in Listing 7 from Section 5.2.2. An intermediate, simplified query which is sufficient for our purpose is shown in Listing 16.

Listing 16 Simplified combination of the queries from Listings 15 and 7. This version only finds the shortest paths between the desired connect and disconnect timestamps.

```
WITH
datetime("2020-04-26T06:30:17.000[Europe/Vienna]") AS timestamp

MATCH
(a:Tl_Timestamp)-[:NEXT*0..1]->(b:Tl_Timestamp),
(connectTimestamp:Tl_Timestamp)-[:NEXT*0..]->(a),
(b)-[:NEXT*0..]->(disconnectTimestamp:Tl_Timestamp),

(connect:Tl_Event)-[:HAPPENED_AT]->(connectTimestamp),
(connect)-[:IS_TYPE]->(:Tl_Type{name: "connect"}),
(disconnect:Tl_Event)-[:HAPPENED_AT]->(disconnectTimestamp),
(disconnect)-[:IS_TYPE]->(:Tl_Type{name: "disconnect"}),
(connect)-[:HAS_PID]->(:Tl_Process)<-[:HAS_PID]-(disconnect),
p = shortestPath(
15      (connectTimestamp)-[:NEXT*1..]->(disconnectTimestamp)
16 )
17
18 WHERE
19 a.timestamp < timestamp AND timestamp < b.timestamp
20 OR
21 a.timestamp = b.timestamp AND a.timestamp = timestamp
22
23 RETURN p
```

Line 2 just specifies the desired timestamp. The interesting part is found in Lines 5–7 which are responsible for retrieving the best matching timestamp nodes and then exploiting the *NEXT* edges for restricting the possible instances of *connectTimestamp* and *disconnectTimestamp* according to the result of the previous search. At this point, the query has two groups of timestamps left for further processing which are either before *a* for after *b*. The remaining Lines 9–16 are taken from already proposed queries and just restrict the nodes to the corresponding events and Lines 18–21 are taken from Listing 15 and belong to the filtering and matching for *a* and *b*.

Unfortunately, it turned out that combining Lines 5–7 in Listing 16 resulted in the query not terminating in a reasonable time frame. Further experiments showed, that the query could be modified to find a solution in an expectable but still not a reasonable time, if the upper bound for the *NEXT* hops was restricted, for example, by [: NEXT*0..100]. However, the execution time would still increase as the upper bound was increased, effectively rendering the query useless for large environments. For this reason, we did not further extend the tested modification of the *timeline model* for query *Q4*.

A trace of the query with the Cypher command *PROFILE*, using an upper bound of 100 showed that during Line 5 of Listing 16, about 23,000 rows are returned as an intermediate result right before applying the *WHERE* part, narrowing them down to 58 rows. With an upper bound of 200, the same intermediate rows consisted of about 41,000 and again 58 rows after the *WHERE* and with an upper bound of 400, 61,877 intermediate rows were found and again narrowed down to 58 rows in the next step.

However, according to *Neo4J Browser*, during these experiments, only 508 events were represented in the database (only a small subset of the data used during performance measurements) and 1,295 elements contained in the database altogether, arising the question, about the content of the vast amount of intermediate rows. In any case, the number of intermediate results and the execution time increased if the upper bound was increased and consequently, the query becomes unusable for large environments in which an upper bound of say 400 is not large enough. Without an upper bound, there seem to be too many paths which need to be traversed to find the results for the query.

9. Conclusion and outlook

In this paper, we proposed three different models for representing timestamps in graph databases. These models were filled with test data originating from a live mail server and we developed queries to retrieve certain predefined information useful during log analysis. We benchmarked our resulting queries against each other according to their complexity, their query performance and by properties about the quality of the models.

In the beginning, the *hierarchical model* appeared to be the most promising model because of its usage of graph database specific structures and it allows indicating the precision of the data. Furthermore, variations of this model were proposed several times in non-scientific sources (Bachman, 2013; GraphGrid, 2015). In fact, it turned out that this model leads to very complex and inscrutable queries.

Another main finding was, that to fully use a graph database's features, some queries require temporary structures which must not persist in the database. This was especially a problem of the *timeline model* because if an arbitrary timestamp was needed for a query, which was not present in the database, the query needed an extension or the overall query had to fall back to a query style like for the *property-based model*. However, an extension of the queries finding fitting nodes for an arbitrary timestamp in the database was theoretically possible, but not practically usable due to excessive execution times.

Based on this information, until the query optimizer can improve the proposed queries good enough, the best solution for storing timestamp information in graph databases is the *property-based model*, as it was the fastest and easiest of all models.

The research results presented in this work will further be used in the author's ongoing research about detecting and mitigating security issues by analysing log files and matching them against the infrastructure topology with the help of graph databases.

Additionally, the resulting models and methods could be applied in the area of cyber physical systems (Auer *et al.*, 2019) or be useful for earlier research of the authors', e.g. using the time-modelling on secure token-based communications for authentication and authorization servers, see (Kubovy *et al.*, 2016).

Notes

1. https://neo4j.com/
2. Native graph databases are characterized by implementing *ad hoc* data structures and indexes for storing and querying graphs.
3. https://github.com/neo4j/neo4j-browser

References

Aichhorn, A., Etzlinger, B., Mayrhofer, R. and Springer, A. (2017), "Accurate clock synchronization for power systems protection devices over packet switched networks", *Computer Science - Research and Development*, Vol. 32 Nos 1/2, pp. 147-158.

Auer, D., Jäger, M. and Küng, J. (2019), "Linking trust to cyber-physical systems", *International Conference on Database and Expert Systems Applications*, Springer, pp. 119-128.

Bachman, M. (2013), "Graphaware neo4j timetree", available at: https://github.com/graphaware/neo4j-timetree

Chazelas, S. (2020), "Pid reuse possibility in linux", available at: https://unix.stackexchange.com/a/414974

Chu, Z., Yu, J. and Hamdulla, A. (2020), "A novel deep learning method for query task execution time prediction in graph database", Future Gen. Comp. Systems.

Cnicutar (2020), "Linux pid recycling", available at: https://stackoverflow.com/a/11323428/8428364

Diederichsen, L., Choo, K.K.R. and Le-Khac, N.A. (2019), "A graph database-based approach to analyze network log files", in Liu, J.K. and Huang, X. (Eds), *Network and System Security*, Springer International Publishing, Cham, pp. 53-73.

GraphGrid, I. (2015), "Modeling time series data with neo4j", available at: www.graphgrid.com/modeling-time-series-data-with-neo4j/

Grüninger, M. (2011), "Verification of the owl-time ontology", *International Semantic Web Conference*, Springer, pp. 225-240.

Hobbs, J.R. and Pan, F. (2013), "Time ontology in owl", available at: www.w3.org/TR/owl-time/

Hofer, D., Jäger, M., Mohamed, A. and Küng, J. (2020), "On applying graph database time models for security log analysis", in Dang, T.K., Küng, J., Takizawa, M. and Chung, T.M. (Eds), *Future Data and Security Engineering*, Springer International Publishing, Cham, pp. 87-107.

Kubovy, J., Huber, C., Jäger, M. and Küng, J. (2016), "A secure token-based communication for authentication and authorization servers", *International Conference on Future Data and Security Engineering*, Springer, pp. 237-250.

Maduako, I. and Wachowicz, M. (2019), "A space-time varying graph for modelling places and events in a network", *International Journal of Geographical Information Science*, Vol. 33 No. 10, pp. 1915-1935.

Neo4j, I (2020a), "2.10. Temporal (date/time) values", available at: https://neo4j.com/docs/cypher-manual/current/syntax/temporal/

Neo4j, I (2020b), "7.5. Shortest path planning", available at: https://neo4j.com/docs/cypher-manual/current/execution-plans/shortestpath-planning/

Patton, A.J. (2001), "Modelling time-varying exchange rate dependence using the conditional copula", SSRN.

Schwenk, J. (2014), "Modelling time for authenticated key exchange protocols", *European Symposium on Research in Computer Security*, Springer, pp. 277-294.

Semertzidis, K. and Pitoura, E. (2016), "Time traveling in graphs using a graph database", *EDBT/ICDT Workshops*.

Tao, X., Liu, Y., Zhao, F., Yang, C. and Wang, Y. (2018), "Graph database-based network security situation awareness data storage method", *EURASIP Journal on Wireless Communications and Networking*, Vol. 2018 No. 1, p. 294.

Theodoulidis, C.I. and Loucopoulos, P. (1991), "The time dimension in conceptual modelling", *Information Systems*, Vol. 16 No. 3, pp. 273-300.

Wiese, B. and Omlin, C. (2009), "Credit card transactions, fraud detection, and machine learning: modelling time with lstm recurrent neural networks", *Innovations in Neural Information Paradigms and Applications*, Springer, pp. 231-268.

Corresponding author
Daniel Hofer can be contacted at: daniel.hofer@jku.at