Efficient evaluation of temporal queries on large graphs

Manders, F.A.M.

Award date:
2020

Efficient evaluation of temporal queries on large graphs

FRANS MANDERS, TU/e

The information needs in the analysis of criminal networks are evaluated and modeled using a framework of three predicates. This thesis discusses the modeling power of this framework in this setting, as well as the implementation in a custom database engine. The performance is evaluated using three distinct indexing techniques and compared against the performance of existing products.

ACM Reference Format:
Frans Manders. 2019. Efficient evaluation of temporal queries on large graphs. 1, 1 (December 2019), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The relational database model is a popular choice for general purpose databases. Its advantages are the well-structuredness and the SQL query language in which tables behave as a first-class data type. Its popularity has led to the integration of range of specialized functionality extensions into the standard.

A graph-based approach is more convenient for solving certain classes of practical problems in which the traversal of interactions, information, or goods between entities is important. Using the knowledge that there is a graph structure makes it possible to allow optimizations for these problems, which are not possible in a relational database. This includes both optimizations in syntax to make the queries more readable, and optimizations in query execution speed. One common model for graph databases is the property graph model. In this model each of the entities and interactions can be extended with key-value pairs.

In the early 90's it was already suggested[20] that graph analysis of criminal activities could improve the quality of work performed by criminal investigators. Since then, much more data has become available, and many research fields have contributed to improve the techniques. Examples are new measures for centrality and the automatic detection of deceptive identities.

There are different methods to model temporal graphs in databases. Traditionally, traversals are defined only by their source and target. This means that the time information is removed, and several actual traversals with time information are aggregated into traversals without time information. Another approach is to consider the traversals to be defined by the source, target, and time on which the traversal occurs. If the times of the traversals are important, the analysis becomes more complex. The temporal relation between traversals, the (temporary) disappearance, the ordering and causality of traversals can be analyzed. The temporal dimension can be considered in the same manner as any other attribute. Using a database specialized for this task instead may result in a performance gain, since again optimizations specific for these problems can be used.

The results of a criminal investigation often must be presented in court. It is important to document the decisions and steps leading to the results. For analyses, such as performed for the *Dover*-case[16], a telecom analysis is performed to determine the locations of suspects. Such an analysis can be extended to a graph analysis by also considering the interactions between persons.

A criminal network poses new challenges which do not apply for social networks in general. For criminal networks, it can be difficult to retrieve all interesting interactions, and the dataset may contain non-relevant interactions. Failing to classify these correctly will result in erroneous conclusions and therefore unusable results.

During the analysis itself, analysts are performing a task such as identifying the roles of the involved persons, or proving whether the persons participate in a *crimineel samenwerkingsverband* (criminal partnership).

After performing several iterations of an analysis, a better understanding of the dataset is gained. This could result in adjusting the parameters in earlier steps of the analysis. Slightly different variations of the analyses must be performed repeatedly on slightly different variations of the preprocessed datasets. Since the analysis task must be performed often, a well performing underlying database engine which can be used interactively and produces repeatable results, is important.

In a police investigation, a criminal network is a network of persons who have knowledge, motivation, and access to resources to perform specific illegal activities[1]. Because we focus on the interactions between these persons, we will modify the definition of a *criminal network* to be the collection of interactions between *entities*.

An *entity* is an identifier, such as a phone number or an email address, belonging to a person in a criminal network as defined by the police. A person in the network has 0 or more different identifiers, and identifiers may be shared or transferred between persons.

An edge in a criminal network corresponds with an interaction between entities. It is defined by, and can therefore be represented by the source, target and timestamp of the interaction. A *timestamp* is a point in time. The only assumptions made about the time representation are that it can only increase and not decrease, and that it can be denoted by an integer. Without these assumptions it cannot be stated that if $timestamp(a) > timestamp(b)$, a happens after b.

The criminal network is modeled as a temporal graph. There are different methods for representing a temporal graph, as discussed earlier. To maximize the usefulness of the data, an edge in the graph must consist of at least a source node identifier, a target node identifer, a start time and an end time for which the edge is valid. The source and target nodes are represented by an unique identifer. Both

Author's address: Frans MandersTU/e, f.a.m.manders@student.tue.nl.

XXXX-XXXX/2019/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://thesaurus.politieacademie.nl/Thesaurus/Term/11354

the start and end time are represented by integers. Another advantage of this format is that it is identical to the format of the input graph.

The main problem is to find a method for efficient retrieval of queries, with emphasis on the queries which are important in a criminal analysis setting. Queries are defined by path predicates, which are functions providing a mapping from candidate results to the boolean values *true* or *false*.

The suitability of this method will be analyzed experimentally with help from experts who are working professionally with existing solutions. This includes a comparison with methods which are currently in use.

This method should perform better than an equivalent implementation of the queries in both *Neo4j* and *PostgreSQL*. Neo4j is currently in use by experts and is specifically designed for querying a graph database. PostgreSQL is one of the major RDMS implementations in use and is well optimized for general types of queries. Since each problem that can be represented in either the relational model or the property graph model can also be represented in the other, both approaches are suitable.

We want to improve the evaluation time by introducing both a framework for formalizing the query in terms of predicates and by analyzing how the query plan of alternative solutions can be improved. Three index structures are compared by their performance.

The hypothesis is that the current state of the art in graph and relational databases could be optimized better for these specific types of queries.

Therefore the research questions are:

(1) Which queries are important for the analysis of criminal social networks with temporal information?
(2) Are current database solutions fast enough to query criminal social networks with temporal information for interactive use?
(3) How can database engines be improved to better handle criminal social networks with temporal information?
    (a) How should predicates used for querying criminal social networks with temporal information be modeled?
    (b) How can indexing techniques be improved to allow faster retrieval of results?

The contributions in this paper are the following. To determine the queries which are important in a criminal analysis setting, field experts were interviewed as shown in Section 3.4. Existing solutions are examined in Section 3.2 to determine whether these are suitable. A framework has been developed in Section 3 to model the queries in a criminal analysis setting. These queries can be modeled within this framework and answered by a database engine. Three different indexing techniques (a topological index, a temporal index and a combined index) have been compared experimentally in Section 5. Finally, suggestions for further research are given in Section 6.1.

2 RELATED WORK

Databases can specialize their use of time data using different methods. Database engines such as *kdb* and *m3db*, which are specialized in handling temporal data in the form of time series have been developed to handle datasets in which a small number of variables change over time. These are specialized in handling large insertion volume of incoming data, and often use column-based storage to handle monitoring and financial transactions. Other databases are specialized in handling the temporal relation between the individual records.

As opposed to the relational database setting, where SQL has been the most popular query language for many decades, there are multiple popular languages for querying graph databases. The most widely adopted is Cypher, but GraphQL and SPARQL are also popular. A formalization of a more expressive query language for graph databases using the property graph model is given in [3]. Of these, SPARQL is the oldest language, and temporal extensions have been developed for it[11].

There is a rich history of research focusing on using databases to help criminal investigations[6]. As an example, crime scripts can be used to predict properties of a crime network.

On the other hand, there is research focusing on the temporal properties of social networks. Temporal k-cliques can be used to query for the time windows for which a temporal k-clique, a set of mutually overlapping interactions, exists[26]. A method for counting topological patterns in temporal graphs is discussed in[18].

In 1988 Gadia proposed a model to add support for temporal data to relational databases[8]. After database vendors such as Postgres and DB/2 had implemented support for temporal datatypes, the data types DATE, TIME, TIMESTAMP, and INTERVAL became standardized in the SQL 92 standard. Because these were standardized after the first implementations, and because of the difficulty of handling many different edge cases, the supported ranges for the datatypes, the operations and the syntax vary per product.

Temporal extensions such as TSQL2 have been developed[19] to extend the SQL 92 functionality to include support for application time. These extensions allow more concise queries, and more efficient querying of the database.

Ideas and concepts, such as temporal operations for inserting, updating and deleting a value at a point or an interval in history (*application time*) are based on the TSQL2 extensions. The possibility of recording when values were written to the database (*system time*) and keeping track of the history is also derived from TSQL2. The introduced concept of bitemporal relations keeps track of when facts were written to the database, and the period for which the facts were true in the real world. This creates new challenges, such as multiple rows having the same primary key, and temporal update queries which affect values for only a part of the period defined by other rows[14]. These standards are being implemented by the major RDBMS's.

Currently, work is being done on implementing temporal joins on time intervals. Two tuples can be joined if they are valid for the same application time. Temporal joins add a new dimension to joins. First it must be decided whether a snapshot or all the edges existing over time are considered. Next, it can be useful to join tuples for intervals which satisfy some condition, such as overlap, or almost overlap of each other[21], or any of the other possible relations between temporal intervals as defined in Allen's interval algebra[1]. Current research in the area of temporal relational data include the efficient evaluation of the k facts which are most often true[9], and data types for specialized use cases.

For fast retrieval of results, general purpose (relational) databases primarily make use of proven techniques such as B-trees[24] or variants for use in parallel systems[15], and hash indexes[25]. The use of B-trees in a graph database requires multiple lookups for paths. Alternatives such as join indexes[22] and Patricia tries[5] which index possible paths which can be traversed are discussed in research literature.

Specialized indexes for temporal data are the multiversion B-tree[2], the timeline index[13] and the bucket index[4]. The multiversion B-tree considers edges as defined by their endpoints. An edge can be enabled or disabled. The index works with versions. A new version is created if the current state of the edges is modified, and each version represents a point in time at which the set of enabled edges changes. A multiversion B-tree consists of multiple B-tree structures. One structure is used to store all edges which have existed at some point in time. A distinct set of B-trees is used to denote which edges are active at a given point in time. Each of the B-trees in the set denotes a distinct interval of the time domain. This index performs well if the user wants to update the current state of the database.

A timeline index also considers edges as defined by their endpoints. Time is modeled using the instants at which edges are activated or deactivated. This makes it possible to model the set of all edges as a sequence of enabling and disabling of edges. In order to retrieve the edges which are activated at a query point, the set of active edges can be reconstructed by traversing the sequence of activations and deactivations. This process can be sped up by storing intermediate checkpoints. Similar to multiversion B-trees, this index performs well when the user wants to update the current state of the database.

The bucket interval index takes a different approach. It combines the sorted index and adjacency index used in regular graph databases and is therefore effectively a composite index. It splits the time domain into time partitions, and each of these partitions has its own adjacency index. This allows a single lookup combining both the temporal and the adjacency properties of a query, and therefore restricting the search space in two dimensions per step.

It is also possible to build an index on top of the abstraction layer of the database, such as the RI-tree does on top of the SQL layer as described in [7].

3 METHODOLOGY

In order to obtain the results for the criminal investigation, the investigators use an online analytical processing engine. As opposed to general purpose database engines, updates or removal of data cannot occur. This simplifies the design, and ensures that the ACID requirements[12] trivially hold.

Since the input graph is defined by its interactions, all entities in the network belong to at least one interaction and can therefore be retrieved by one of their incident interactions. It is not possible to model entities which are not interacting with others. It is therefore suitable to define the graph using only edges.

Candidates are n-tuples of interactions. In order to obtain the results from the dataset and the query, the result candidates need to be filtered by predicates. A predicate is a mapping from a result candidate to the boolean value *true* if the result candidate satisfies the predicate or *false* if it is not.

We will consider three distinct types of path predicates: a temporal predicate P_{temp}, a topological predicate P_{top}, and a general predicate P_{gen}. A result is a result candidate for which all three predicates return *true*.

P_{temp} is restricted to a conjunction of the boolean functions defined using only the values of $tstart$ and $tend$ of each edge as variables, a constant, and the comparison operators $=$, $>$, $<$, $\geq$, and $\leq$.

The P_{top} is restricted to the conjunction of boolean functions using only the values of the $source$ and $target$ of edges, and constants. There is no order defined over the edges, so only the comparison operator $=$ can be used. The topological predicate encodes the result length, and exactly all edges used in the predicate body must be mapped to an edge in the result. The edges do not need to be distinct.

The general predicate P_{gen} is not restricted in the properties of the edge it can contain, it only needs to be possible to be implemented in a deterministic program. This means that it can contain $\neq$ in addition to the operators available for the other predicates. The general predicate is added to allow searching for results consisting of two interactions having the same source but a distinct endpoint. It can also be used for many other filtering predicates, such as filtering on interactions occurring on the same day each week.

The results for each query are obtained by collecting all result candidates rc for which the following holds:

$$\{results = rc \mid P_{temp}(rc) \wedge P_{top}(rc) \wedge P_{gen}(rc)\}$$

The result set can be obtained by evaluating the predicates over all possible combinations of edges in the dataset. If only one of the predicates returns *false*, it is certain that the result candidate does not belong to the result set.

A stricter definition of the predicates, for example restricting the predicates to operate only on the values of a single edge, is useful for application in distributed systems. This makes independent evaluation of the filters possible, so that the engine can be scaled horizontally to multiple machines. Another advantage of this is that it allows more effective caching of intermediate results if a query filters more than one interaction in a pattern using the same predicates.

Conceptually, the results are the elements generated by the enumeration of all the result candidates, for which all filters return *true*. A more realistic alternative for the expensive enumeration of the result candidates is to skip over results which are known not to satisfy one of the predicates. A topological index can be used to skip result candidates which do not have the correct topological structure, and a sorted index can be used to skip result candidates which do not satisfy the temporal constraints. No index will be implemented for the general predicate, since the lack of restrictions allows no general strategy to skip over non-results.

The result candidates are n-sized tuples of the form

$$\langle edge_1, edge_2, \ldots, edge_n \rangle$$

If the user queries for a path of length 6 between entity a and entity b, the result candidates are all tuples of 6 edges, and the results are

the tuples of 6 edges denoting a path between a and b, satisfying all three predicates of the query.

In the property graph model, each node or edge can be extended with a set of property keys. Each property key has an associated value. This makes this model convenient to model interactions with properties, such as the starting and ending time. This research focuses on the temporal properties and the entities in the result candidates, but other properties, such as the contents, the language of the interactions, or the importance of the entity may be added to the graph and queries.

In the relational model, the SQL standard defines methods for representing system time intervals and application time intervals for information stored in a table, and operations over temporal intervals. How much is implemented in relational databases and the terminology used varies per product. To model the queries for this analysis, only the application time is of interest. PostgreSQL for example uses the AS OF syntax allows to query which information was valid at a specific point in time, and operators for checking whether two intervals overlap (&&) or whether an interval contains an instant (@>) are available[23]. However, these are not defined in the SQL standard and are therefore nonportable and subject to change. Defining a set of columns as defining the application time is therefore equivalent to creating a B-tree index on these columns. Using these vendor-specific constructs provides no additional gains and makes the model less portable. Therefore, integer columns will be used to represent the interval for which the facts are valid.

Lookups for the topological structures in a relational database are most efficiently performed using a hash index. This requires fewer lookups than when using a B-tree index.

The result candidates can be represented as a collection of edge tuples in the property graph model.

The method chosen to ensure efficient retrieval of queries is to determine an efficient index structure. In order to do this, an edge e_i in the property graph model is encoded as follows:

$$e_i = \langle i, source_i, target_i, time\_start_i, time\_end_i \rangle$$

This can be obtained from the edges in the dataset by using the values $e_i = \langle i, source_i, target_i, time_i, time_i \rangle$. The encoding is therefore more general than the format of the input dataset, and a larger problem space can be modeled.

To determine the efficiency, the data is loaded both in the existing solution and in our custom engine. Query times are measured and compared. The duration of creating the index is not considered to be important. This operation is performed much less often and can be performed in the background. Additionally, the time required to generate the indexes is much less than for the nontrivial queries.

The experiment results are reported to determine whether the solution provides sufficient performance. Evaluation durations are measured in milliseconds, and 9 durations are reported for each experiment. These are summarized by reporting the first measurement to account for an empty cache and the median of the measurements to account for outliers. When querying a similar query multiple times, it is expected that caching of intermediate results will significantly improve the performance after the first measurement.

The queries have been reduced to a combination of three different types of filtering queries, which can capture most of the above queries.

(1) Queries for a topological structure. Example: Find edges a, b, c such that $target(a) = source(b) \wedge target(b) = source(c)$.
(2) Queries in time. Example: Find edges a such that $10\,000 \leq time\_start(a) \leq time\_end(a) \leq 20\,000$.
(3) Queries involving any property of entities and interactions, with any type of deterministic constraint.

Since this information can be captured in a simple structure, and the queries are sufficiently expressive to encode most of the queries it is expected that this choice is a good balance between performance and expressiveness to be of practical use.

These combination of query types are found to be representative for the information needs in the investigation of a criminal network.

3.1 Aggregation

It is expected that the performance of the benchmarks for our implementation, as well as for the relational model and the property graph model vary depending on the distribution of the time windows and the number of elements in the dataset. The obtained dataset has recorded interactions between entities as instants. Since there are many natural ways to aggregate this data into a new dataset which makes use of time windows, two distinct methods have been chosen.

- Aggregate edges from a given a to a given b in 1-week intervals. The earliest edge e not yet assigned to an interval is assigned to a new interval starting at $time(e)$. All edges f with $time(f) \leq time(e) + 1\ week$ are ignored, and the process is repeated. This results in approximately 66 000 edges, distributed as in Figure 3.
- Aggregate all edges from a given a to b into a single edge. Only a single edge from a to b remains. This results in approximately 22 000 edges which are distributed as in Figure 2.

3.2 Baseline implementations

A baseline comparison has been performed by comparing the performance against both an SQL query in PostgreSQL 11 and a Cypher query in Neo4j 3.5.13. Both are the latest stable release of the engine at the moment of writing. The engines are used with the default settings after installation.

Both preprocessed datasets have been inserted into both a PostgreSQL and a Neo4j database. For the PostgreSQL implementation, hash indexes have been created for the source and target nodes, and B^*-tree indexes have been generated for the *start_time* and *end_time*. These index types are typical for a relational database.

In the Neo4j implementation, the default adjacency indexes have been generated, and B+-tree indexes for the time properties of the edges.

In both baseline implementations, the data has been inserted using a bulk insertion method. To ensure a similar state of the database before the execution of each analysis query, a warm-up query which loads all the data into memory is performed. For PostgreSQL this is the function *pg_prewarm*, for Neo4j this is a manual query which loads all entities, relations and their properties. This has no clear effect on the query times for the PostgreSQL database.

It was considered to wrap the analytics queries in a `count` call to minimize the impact of the transfer time of the data to the client. However, PostgreSQL does not require access to the actual tuples, the index is sufficient[25]. This makes these results not representative for the queries without a `count` aggregation. The queries use the wildcard select filter (`select *`) to force the engine to retrieve all fields of a tuple.

3.3 Querying

The index structures are first generated for each dataset. After this, each of the queries is evaluated 9 times, except for the long running ones, which are evaluated just once. The queries are terminated manually when it has become apparent which of the index methods is most suitable.

Two different types of indexes are compared. A B-tree for fast range lookups for the time values, and an adjacency index for fast equality lookups for filtering topological structures. The indexes are generated before the evaluation of the queries.

It is expected that using the B-tree index will make queries using a very restrictive temporal predicate faster (such as Q7), and that using an adjacency index will make evaluation of queries using a very restrictive topological structure (such as Q3) more efficient.

The typical query involves a topological component, a temporal component, and a general component. Each of these components can be represented by a predicate. In the most general form, such predicate is defined by a function, which returns *true* if the result candidate matches, and *false* if it does not.

For the topological predicate, an alternative representation is a list of pairs. Each pair consists of two variables. The n-th pair in the list represents the n-th edge of the result candidate. The first variable of a pair represents the number for the source of the edge, the second variable of the pair represents the target of the edge. A variable must be mapped to exactly one node. Therefore, if a variable appears more than once in the predicate, the node must appear more than once in the result. This method makes it possible to only denote motifs of a fixed length. Additionally, a default initial mapping can be given. This makes it possible to search for paths starting at a given node.

Each temporal predicate is defined by a left-hand value, a comparison operator and a right-hand value. Both the left-hand value and the right-hand value may refer to the start time of an edge, the end time of an edge, or a constant value. The comparison operator can be *equal to*, *unequal to*, *less than*, *strictly less than*, *greater than*, *strictly greater than*.

The combinations of all temporal predicates are stored in a list. This list of temporal predicates can be used to represent all possible cases described in Allen's interval algebra[1], and its extension to multiple edges.

The general predicate can be anything that can be written as a function of the candidate result. Because it allows so much freedom, it cannot be used to look up values in an index.

9 measurements are performed for each combination of index strategy (or baseline implementation) and dataset. A first measurement is stopped after 3 hours, or 2 hours on the social criminal networks because of the limited availability of the computers. The remaining 8 measurements are only performed if the first measurement takes 20 seconds or less. This limit is chosen as an upper bound of the approximate 10 second boundary for allowing continuity thinking[17].

3.4 Interview

During several interviews and feedback from both the NFI as the Police it was determined that the queries which are interesting in the investigation of a social network of criminals are the following:

The following information needs have been determined by both the police and the NFI to encode the required information in the analysis of a criminal network:

(1) Retrieve the persons having an interaction at a given time, or in a time interval.
    This question can be asked in a murder investigation. In order to answer this question, the interacting entities must be retrieved, and these need to be linked to persons.
(2) Retrieve the pairs of entities which interact often with each other in a particular time window.
    Entities which interact often may be interesting from the criminal partnership setting.
(3) Retrieve the entities that could have transferred information to a user within a time interval.
    Currently, this is not seen often in practice according to the police.
(4) Retrieve the distinct times on which a user had an interaction.
    This is usually not done in combination with a social network analysis, but with tap data only.
(5) Retrieve the common neighbors of any two entities.
    This is asked often, especially in investigations involving illegal drugs.
(6) Compute the degree of all entities in a graph.
    Currently, the datasets are usually too large to perform this calculation and programs used for analysis crash.
(7) Which entities are similar?
    This can be analyzed from different perspectives, for example searching for patterns in the graph.

Additionally, queries which are not used at the moment, but which could be useful and can be modeled using the framework are:

(1) Retrieve the entitites that could have transferred information to a user in a time interval

During the interview, no concrete limits on the query evaluation time were found. However, it was found that most of the queries are performed interactively. This makes a query time in the order of seconds, instead of minutes, hours or longer, desirable.

4 IMPLEMENTATION

Different indexing strategies were considered for implementation. Multiversion B-trees are difficult to implement, a timeline index does not provide a suitable performance because it either must scan the entire set of interactions for each node expansion if few checkpoints are created, or it is not able make good use of the cache if many checkpoints are created. The RI-tree is only an artificial layer on top of the real index, so any performance gain over other databases can be further improved by implementing the index directly. To be able

to compare three distinct indexes, a topological index, a temporal index, and a combined index have been implemented in a custom database engine.

The preprocessing steps to reformat and aggregate the data are implemented in Python. The engine itself is implemented in C++. The results are retrieved in a pull-based manner, such as described in [10].

For the experiments, the results are constructed using a depth-first search over one of the indexes. For temporal predicates, a lookup in the B-tree index for temporal relations is performed. For topological predicates, a lookup in the adjacency index for topological relations is performed. Both predicates can be combined in a bucket index. If a certain index cannot be used for a particular predicate, a full scan over all tuples will be performed.

The predicates are implemented in C++ code. The topological predicate is implemented as a list of variables for $source, target$. Additionally, an incomplete mapping from variables to node id's is given. The predicate evaluates to *true* for a result candidate iff every variable can be mapped to a node id such that the list of variables represents the motif of the query candidate. The given incomplete mapping from variables to node id's must be a subset of the mapping for the complete mapping.

The temporal predicate is implemented as a list of equations. Each equation involves a lhs, a comparison operator, and a rhs. Both the lhs and the rhs can be either a constant temporal value, or a the $tstart$ or $tend$ value of an edge in the result candidate. The comparison operator can be one of $\leq, <, =, >, \geq$. The temporal predicate evaluates to *true* if all equations of the form $lhs(comparisonoperator)rhs$ evaluates to *true*.

The general predicate is implemented as a C++ function which accepts a result candidate as an argument. It evaluates to *true* if the C++ function returns *true*.

It is convenient to encode predicates such that it holds that $P(edge_1, \ldots, edge_k) = false$ if $P(edge_1, \ldots, edge_k, edge_{k+1}) = false$ for all possible $edge_{k+1}$ for any k. If this is done, the depth-first search can backtrack early and skip many non-results early.

A B^*-tree index is generated using an implementation provided by Google[2]. B-trees and its variants are more sophisticated improvements over sorted lists of data and can quickly traverse over all matching tuples within an interval. Because it also considers the page size, it is very efficient when updating the tree with new values. Random access time and insertion times are improved, and the performance is predictable. This type of index is popular in both relational databases and graph databases for range lookups in a continuous domain.

The topological index has been implemented as a clustered index using hash maps and arrays in C++. Storing the index together with the interactions is more efficient in space and processing time than storing it together with references to the data. The disadvantage that it becomes more costly to update. However, this does not apply for this type of database, which does not support updates. This type of index quickly performs equality lookups and is in some databases implemented as a hash lookup.

The combined index has been generated by splitting the temporal domain in 3 partitions. For each partition a topological index has been constructed to perform the lookups for that partition. Before the predicates are evaluated over the result candidates, the duplicate result candidates for each partition are removed.

5 EXPERIMENTS & DISCUSSION

A dataset of interactions that was used for a criminal investigation has been made available by the NFI for validation. The interactions were obtained as a part of a criminal investigation of a criminal network and form interactions between entities, which can be thoughts of as e-mails.

For the PostgreSQL comparison, a hash index has been generated for the *source* and *target* of each interaction. A B-tree has been generated for the *tstart* and *tend* values. Additionally, a composite index over the *source*, *target* and *tstart* column has been created. PostgreSQL is allowed to determine the optimal index to use.

The dataset that has been made available is not the original dataset that was used for the investigation. Identities have been replaced by pseudonyms, and the times are shifted by an unknown amount, to ensure that the data cannot be linked to the original case. Neither of the transformations influences the performance measurements.

The dataset contains approximately 500 000 edges e_i represented as follows:

$$e_i = \langle source_i, target_i, time_i \rangle$$

There are 19 922 entities in the network, if both sources and targets of the interactions are counted. The time has a resolution of seconds. Only an integer representing the number of seconds since some unknown, but constant point in time are known.

The time difference between the first recorded timestamp and the last recorded timestamp is 90 007 831 seconds (approximately 3 years).

An example of a tuple in the dataset is:

$$edge = \langle entity_1, entity_2, 448371555 \rangle$$

The instants in the dataset are distributed as shown in Figure 1.

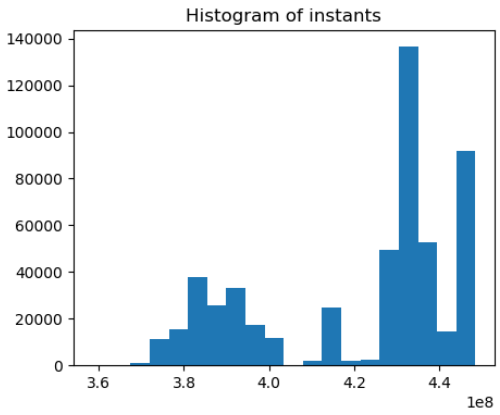

Fig. 1. Distribution of instants in the dataset

[2]https://isocpp.org/blog/2013/02/b-tree-containers-from-google

Q#	PostgreSQL		Neo4j		Adjacency first		Temporal first		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	7	0	499	32	4	1	8	7	35
2	14	0	82	25	5	2	8	8	233
3	1	1	346	13	361	1	$> 45min$	9	544119
4	0	1	44	14	5	1	13	9	17991
5	0	0	106	23	1	1	9	9	0
6	0	0	19	9	1	1	9	9	3
7	$> 2hours$		$> 2hours$		$> 2hours$	$> 2hours$	$> 2hours$	$> 2hours$	?
1a					286	1	25	7	281
2a					151	1	14219	8	742
3a					526831	*	$> 2hours$		286542169
4a					2230	1			1567879

Table 1. Measurement times in milliseconds in reference implementations on the single interval dataset. *) Only a single measurement was performed.

Q#	PostgreSQL		Neo4j		Adjacency first		Temporal first		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	55	54	671	72	14	4	24	18	13079
2	142	142	147	54	14	4	28	26	65350
3	2	3	356	11	5275	5	$> 2hours$		103
4	0	0	53	12	14	5	39	29	8
5	1	1	66	12	4	5	26	26	80
6	0	0	27	3	4	4	26	27	12
7	794	805	$> 2hours$			4	$> 2hours$		919
1a					51	10	18	18	7477448
2a					250528		$> 2hours$		10505937
3a							$> 2hours$		1232587
4a									45521

Table 2. Measurement times in milliseconds in reference implementations on the windowed intervals dataset. *) Only a single measurement was performed.

Q#	Adjacency first		Temporal first		Bucket index		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	5	0	20	8	5	1	0
2	2	0	4	4	2	0	0
3	4033	0	145958	NM	6005	1	2698531
4	0	0	9	8	1	1	252
5	12	0	527	8	21	1	31375
6	0	0	8	8	1	1	246
7	1696	0	103	8	2587	1	
1a	7	0	8	8	8	1	0
2a	4	0	4	4	5	1	0
3a	$> 2hours$	NM	$> 5hours$	NM	$> 3days$	NM	
4a	202783	NM	3625471	NM	318519	NM	171552690

Table 3. BostonTrain dataset

The queries can be formalized in terms of the interactions as follows:

(1) $A = \{e \mid t\_start(e) \leq t\_start(w) \wedge t\_end(e) \geq t\_end(w)\}$

(2) Retrieve $E = \{e \mid (t\_start(e) \geq t\_start(w) \wedge t\_end(e) \leq t\_end(w)) \wedge (t\_start(e) \neq t\_end(e))\}$, and count the number distinct combinations of $\langle source(e), target(e)\rangle$.

Q#	Adjacency first		Temporal first		Bucket index		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	2	2	3	0	2	0	0
2	1	1	1	0	1	0	0
3	0	3	92	0	1	0	211
4	0	3	3	0	0	0	6
5	0	3	35	0	1	0	851
6	0	3	3	0	0	0	63
7	51	3	13	0	73	0	0
1a	3	3	3	0	3	0	0
2a	2	1	1	0	2	0	0
3a	465353	NM	$> 3hours$	NM	532742	NM	64297280
4a	1005	NM	48446	0	1432	0	759999

Table 4. ChicagoBike dataset

Q#	Adjacency first		Temporal first		Bucket index		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	258	54	451	462	266	64	0
2	166	54	280	281	174	64	0
3	11173957	NM	$> 4days$	NM	17520034	NM	0
4	45	43	459	463	52	43	2258
5	1799	42	440982	NM	3094	42	4688160
6	44	42	460	463	52	42	3658
7	1846621	NM	231367	NM	2939880	NM	8897
1a	415	50	464	472	426	50	0
2a	260	50	280	281	270	50	0
3a	$> 3hours$	NM	$> 3hours$	NM	$> 3hours$	NM	
4a	$> 3hours$	NM	$> 3hours$	NM	$> 3hours$	NM	

Table 5. FHV dataset

Q#	Adjacency first		Temporal first		Bucket index		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	6	1	13	12	6	1	0
2	4	1	6	6	4	1	0
3	2961	2	121890	NM	2961	1	2743467
4	0	1	14	463	0	1	69
5	32	1	1735	NM	32	1	81986
6	1	1	11	463	1	1	108
7	165	1	133	NM	1965	1	0
1a	11	1	11	472	11	1	0
2a	6	1	7	281	6	1	0
3a	$> 3hours$	NM	$> 3hours$	NM	$> 3hours$	NM	
4a	283448	NM	8114465	NM	283448	NM	252147178

Table 6. Flight dataset

(3) Retrieve the temporal k-clique A of a walk, which exists within time window w

(4) Retrieve $E = \{\langle e_1, e_2, e_3\rangle \mid target(e_1) = source(e_2) \wedge target(e_2) = source(e_3) \wedge tstart(e_1) \leq tstart(e_2) \wedge tstart(e_1) \leq tstart(e_3) \wedge tend(e_3)\}$

Q#	Adjacency first		Temporal first		Bucket index		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	2	0	105	0	4	0	2100
4	0	0	0	0	0	0	16
5	0	0	5	0	0	0	205
6	0	0	0	0	0	0	38
7	24	0	4	0	34	0	0
1a	0	0	0	0	0	0	0
2a	0	0	0	0	0	0	0
3a	232949	NM	3772424	NM	25424	NM	57251615
4a	235	NM	3396	0	301	0	177596

Table 7. MetroBike dataset

Q#	Adjacency first		Temporal first		Bucket index		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	3	0	5	5	3	0	0
2	2	0	3	3	2	0	0
3	9	0	998	6	12	1	5323
4	0	0	5	5	0	0	24
5	0	0	69	6	1	0	1058
6	0	0	5	5	0	0	6
7	164	0	51	5	228	1	0
1a	5	0	5	5	5	0	0
2a	3	0	3	3	3	0	0
3a	2302789	NM	1992991	NM	NM	NM	393549551
4a	2598	0	227509	NM	4015	NM	2298724

Table 8. NYCBike dataset

Q#	Adjacency first		Temporal first		Bucket index		Cardinality
	Cold run	Hot run	Cold run	Hot run	Cold run	Hot run	
1	77	30	104	101	78	35	13079
2	80	32	145	111	84	35	65350
3	28	29	2339	129	31	32	103
4	27	27	123	123	31	32	8
5	29	27	673	123	31	32	80
6	31	27	123	123	30	32	12
7	2347	32	$> 3hours$	NM	2877	35	919
1a	419	28	$> 3hours$	NM	462	35	45521
2a	42233	NM	$> 3hours$	NM	46826	NM	7477448
3a	80385	NM	$> 3hours$	NM	89999	NM	10505937
4a	2866	30	$> 3hours$	NM	3711	34	1232587

Table 9. SocialNetwork dataset

(5) Retrieve $A = \{target(e_1) \mid a \wedge target(e_1) = source(e_2) \wedge target(e_2) = c \wedge e_1 \in E, e_2 \in E\}$.
(6) Count for each node the number of outgoing edges
(7) Search for an entity that has an indirect interaction (via a single other interaction) to an entity having many (at least 4) interactions

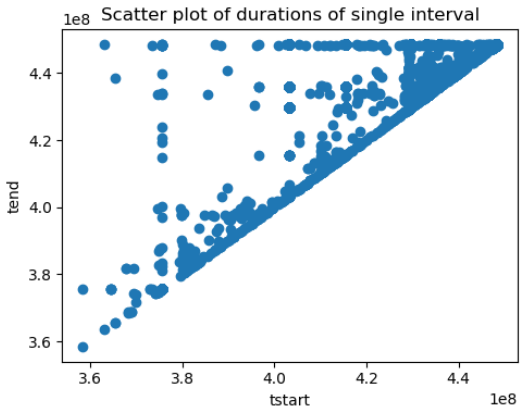

Fig. 2. Distribution of interval start and end times after aggregating by sources and targets

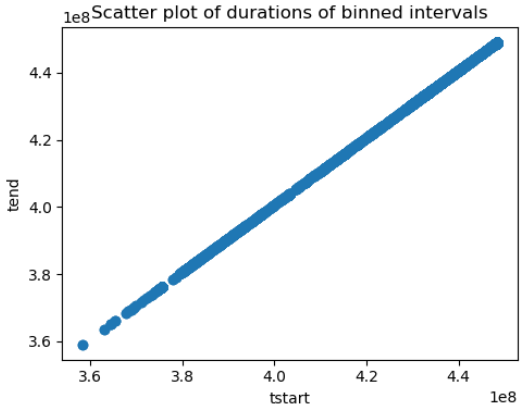

Fig. 3. Distribution of interval start and end times after aggregating by source and targets in weekly windows

Graphical representations of the queries are displayed in Appendix A.

Note that all these queries involving only single interactions can be extended to more general patterns involving more interactions if required for the use case.

Many of the information needs need a slight adjustment before they can be answered using this framework. These require another aggregation step or to be performed repeatedly with different parameters.

In Table 1 the performance measurements for the smaller single interval dataset are summarized.

The performance measurements for the larger windowed dataset are displayed in Table 2. Note that the measured query times for the hot run are much less than the cold run for the adjacency first method. In order to explain the results, the queries were executed again under the Vagrant Cachegrind tool. The tool reports that the *total* number of L1 cache misses is larger if every query is executed only once, instead of 9 times. This difference is therefore cache related.

For both experiments, only a single run is performed if the first run takes longer than 20 seconds. The results of the subsequent runs are not measured. If the first run takes more than 45 minutes (or 2700 000 ms), it is aborted and reported as such. This is also done if the results show that another approach works much better.

Additional experiments have been performed on other datasets. The SocialNetwork dataset is specifically constructed to be similar in structure as the windowed interval dataset. The results of these are listed in Table 3, Table 4, Table 5, Table 6, Table 7, Table 8 and Table 9. The sizes of the datasets are shown in Table 10. The experiments were not measured for the bucket index on the single interval dataset or the windowed intervals dataset. If the results from the experiments on the BostonTrain, ChicagoBike, FHV, Flight, MetroBike, NYCBike, and SocialNetwork dataset are extrapolated, the bucket index is expected to perform worse than the topological index.

All experiments are performed using an Intel i5-6300U dual core processor. The results of the baseline measurements are shown in Table 1 and Table 2. The results are verified using other datasets on a more powerful Intel Xeon X5676.

The memory usage is bounded from above by the size of each interaction multiplied by the number of interactions in the pattern multiplied by the number of interactions in the dataset, plus the size of the index. This means that the memory usage scales linearly with the number of the interactions in the dataset.

While the Neo4j database engine is designed for querying graph databases, PostgreSQL performs much better. So far, the results of the custom designed engine, which performs the topological filtering first is faster than either of the already available solutions.

The caching of the results appears to be a factor for both Neo4j and PostgreSQL, but not as significant as expected. The results vary at most a factor 2 between the first and the median of the expectations for the trivial queries. The results are very consistent for the more complicated queries.

Additionally, the framework used for the queries is mostly sufficiently expressive to retrieve the information required for the desired results, however further processing of the results may be required.

Using a B-tree index for filtering on the temporal dimension first does not appear to gain an advantage in retrieval time, except for a minor change in queries with trivial topological component.

Filtering on the topological component of the result candidates is more useful, especially if the user queries for large topological structures.

The reported memory usage remains below 60 megabytes when using the topological, temporal or bucket index.

The indexing methods have been further analyzed using different datasets. These analyses indicate that the bucket index and the topological index have a much better performance gain between a cold run and a hot run than the temporal index does. One of the major differences between these index approaches is that the topological and bucket index are an in-place index, meaning that the index is not a collection of pointers to tuples stored somewhere in memory, but a collection of tuples. Simulations using Cachegrind

Dataset	Size
BostonTrain	12388
ChicagoBike	4504
FHV	585691
Flight	17009
MetroBike	1185
NYCBike	8178
SocialNetwork	157000

Table 10. Number of tuples in the datasets used for the experiments.

show that the number of CPU L2 cache misses is much lower for the hot runs, suggesting that the data can be cached much more effectively if they are small and in consecutive memory.

6 CONCLUSION

The indexing method makes a big difference on the performance of queries. There is no single indexing strategy which is best for all the analyzed queries. As expected, a temporal index performs better if the bottleneck of a query is the temporal lookup, and an adjacency index performs better if the topological structure of the query is the bottleneck.

Some other desirable properties of indexes have been found. An index must be able to skip over non-results quickly and early in the search tree. This is especially important for large datasets and if the results consist of many tuples.

A more complicated index, even if it can skip over more non-results such as the bucket index, is not in general better than a simpler index such as the adjacency index.

Using an in-place index can speed up the queries in a hot run by more than an order of magnitude, compared to using pointers to reference to the actual tuples. This also makes the query times less consistent than when a B-tree index is used. The in-place index is only suitable if the cost of updating the graph is not important.

6.1 Future work

Performing this research has lead to better insights in both the problem domains. This research opens the way for improving the performance of a subset of these queries on a subset of the indexing methods. Because there are tight constraints on the usage of the real dataset, a more promising direction is to put more emphasis on the generation of datasets with similar properties, so that the results can be better extrapolated.

REFERENCES

[1] James F Allen. 1990. Maintaining knowledge about temporal intervals. In *Readings in qualitative reasoning about physical systems*. Elsevier, 361–372.

[2] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An asymptotically optimal multiversion B-tree. *The VLDB Journal—The International Journal on Very Large Data Bases* 5, 4 (1996), 264–275.

[3] Angela Bonifati, G.H.L. Fletcher, Hannes Voigt, and N. Yakovets. 2018. *Querying graphs*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[4] Panagiotis Bouros and Nikos Mamoulis. 2017. A forward scan based plane sweep algorithm for parallel interval joins. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1346–1357.

[5] Brian F Cooper, Neal Sample, Michael J Franklin, Gisli R Hjaltason, and Moshe Shadmon. 2001. A fast index for semistructured data. In *VLDB*, Vol. 1. 341–350.

[6] P.A.C. Duijn. 2016. Detecting and disrupting criminal networks: A data driven approach. (2016).

[7] Jost Enderle, Matthias Hampel, and Thomas Seidl. 2004. Joining interval data in relational databases. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 683–694.

[8] Shashi K Gadia. 1988. A homogeneous relational model and query languages for temporal databases. *ACM Transactions on Database Systems (TODS)* 13, 4 (1988), 418–448.

[9] Junyang Gao, Pankaj K Agarwal, and Jun Yang. 2018. Durable top-k queries on temporal data. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2223–2235.

[10] Goetz Graefe. 1990. *Encapsulation of parallelism in the Volcano query processing system*. Vol. 19. ACM.

[11] Fabio Grandi. 2010. T-SPARQL: A TSQL2-like Temporal Query Language for RDF. (2010), 21–30.

[12] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)* 15, 4 (1983), 287–317.

[13] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1173–1184.

[14] Krishna Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *ACM Sigmod Record* 41, 3 (2012), 34–43.

[15] Philip L Lehman et al. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[16] Lex Meulenbroek and Paul Poley. 2015. *DNA Match*. De Bezige Bij.

[17] Robert B Miller. 1968. Response time in man-computer conversational transactions.. In *AFIPS Fall Joint Computing Conference (1)*. 267–277.

[18] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM '17)*. ACM, New York, NY, USA, 601–610. https://doi.org/10.1145/3018661.3018731

[19] Richard T Snodgrass. 2012. *The TSQL2 temporal query language*. Springer Science & Business Media.

[20] Malcolm K. Sparrow. 1991. The application of network analysis to criminal intelligence: An assessment of the prospects. *Social Networks* 13, 3 (1991), 251 – 274. https://doi.org/10.1016/0378-8733(91)90008-H

[21] James Terwilliger. 2019. Trill 102: Temporal Joins. https://cloudblogs.microsoft.com/opensource/2019/05/01/trill-102-temporal-joins/.

[22] Patrick Valduriez. 1987. Join indices. *ACM Transactions on Database Systems (TODS)* 12, 2 (1987), 218–246.

[23] Peter Vanroose. 2015. Temporal Data & Time Travel in PostgreSQL. https://wiki.postgresql.org/images/6/64/Fosdem20150130PostgresqlTemporal.pdf.

[24] Hartmut Wedekind. 1974. On the Selection of Access Paths in a Data Base System. *Data Base Management*, 385–398.

[25] Markus Winand. 2012. *SQL performance explained: everything developers need to know about SQL performance;[covers all major SQL databases]*. M. Winand.

[26] Kaijie Zhu, George Fletcher, Nikolay Yakovets, Odysseas Papapetrou, and Yuqing Wu. 2019. Scalable Temporal Clique Enumeration. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases (SSTD '19)*. ACM, New York, NY, USA, 120–129. https://doi.org/10.1145/3340964.3340987

A QUERIES

The queries are represented in both Cypher and SQL. For the custom engine, the queries are implemented using the proposed tri-predicate model.

Q1:

Cypher: `match ()-[a]->() where a.time_start <= 380000000 and a.time_end >= 400000000 return a;`

SQL:

```
select * from binned_intervals_interactions where
tstart <= 398144000 and tend >= 401772800;
```

The predicates of the first query can be represented by

$$P_{temp}(t) = tstart(edge_i) \leq 380000000 \wedge tend(edge_i) \geq 400000000 \; \forall i \in \{0, \ldots, n-1\}$$

and $P_{top}(t) = source(edge_1) = source(edge_1)$ and $P_{gen}(t) = true$, or graphically by

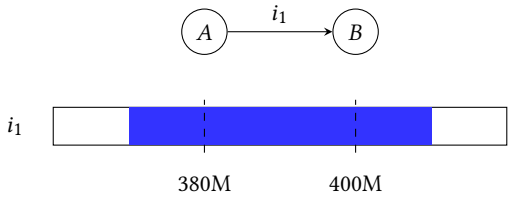

Fig. 4. Q1. The window of the interaction must contain both $380M$ and $400M$.

PostgreSQL evaluates the query by performing a bitmap heap scan.

Q2:

Cypher:

```
match ()-[a]->() where a.time_start >= 400000000 and
a.time_end <= 420000000 return a;
```

SQL:

```
select * from binned_intervals_interactions where
tstart >= 400000000 and tend <= 420000000;
```

and the predicates of the second query by:

$$P_{temp}(t) = tstart(edge_i) \geq 400000000 \wedge tend(edge_i) \leq 420000000 \; \forall i \in \{0, \ldots, n-1\}$$

and $P_{top}(t) = source(edge_1) = source(edge_1)$ and $P_{gen}(t) = true$, or graphically by:

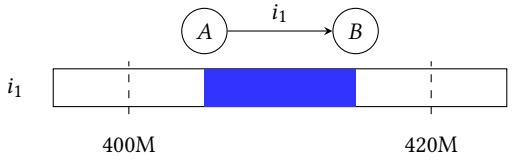

Fig. 5. Q2. The window of the interaction should be between $400M$ and $420M$.

PostgreSQL evaluates the query by performing a sequential scan.

Q3:

Cypher:

```
match (n1)-[e1]->()-[e2]->()-[e3]->(n2) where
n1.id = 1 and n2.id = 15 and
e1.time_start <= e2.time_start and
e1.time_start <= e3.time_start and
e2.time_start <= e3.time_end
return e1, e2, e3;
```

SQL:

```
select * from binned_intervals_interactions as e_1
left join binned_intervals_interactions as e_2 on
e_2.source = e_1.target left join
binned_intervals_interactions as e_3 on e_3.source =
e_2.target where e_1.source = '1' and e_1.tstart <=
e_2.tstart and e_1.tstart <= e_3.tstart and e_2.tstart
<= e_3.tstart;
```

Query 3 can be represented by the predicates $P_{temp} = tstart(edge_1) < tend(edge_2) \wedge tstart(edge_1) < tend(edge_3) \wedge tstart(edge_2) < tend(edge_3)$ and $P_{top} = target(edge_1) = source(edge_2) \wedge target(edge_2) = source(edge_3)$, or geographically by:

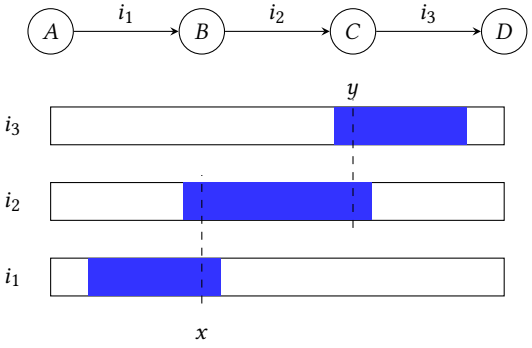

Fig. 6. Q3. The windows of interactions i_1 and i_2 should both include some instant x, and the windows of interactions i_2 and i_3 should both include some instant y.

PostgreSQL evaluates the query by performing a nested loop operation.

Q4:

Cypher:

```
match (n1)-[e1]->() where n1.id=1 return e1
```

In order to determine the distinct timestamps, all timestamps have to be retrieved.

SQL:

```
select * from binned_intervals_interactions where source = '1';
```

Query 4 can be represented by the predicates $P_{temp} = true$ and $P_{top} = source(edge_1) = 1$. Query 4 can also be graphically represented by:

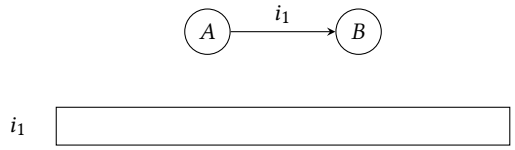

Fig. 7. Q4

PostgreSQL evaluates the query by performing a bitmap heap scan.

Q5:

Cypher:

```
match (n1)-[e1]->()-[e2]->(n2) return e2;
```

SQL:

```
select * from binned_intervals_interactions as
e_1 left join binned_intervals_interactions as
e_2 on e_1.target = e_2.source where e_1.source = '7';
```

Query 5 can be represented using the predicates $P_{top} = target(edge_1) = source(edge_2)$ and $P_{temp} = true$ and also graphically represented by:

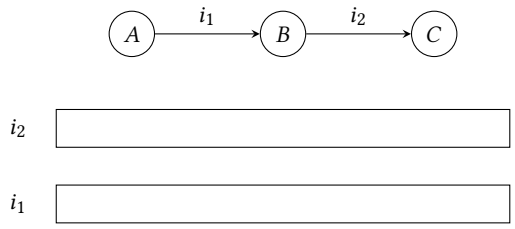

Fig. 8. Q5

PostgreSQL evaluates the query by performing a hash join first.

Q6:

Cypher:

```
match (n1)-[e1]->(n2) return e1
```

SQL:

```
select * from binned_intervals_interactions where source = '13';
```

Query 6 can be represented using the predicates $P_{top} = source(edge_i) = 1$ and $P_{temp} = true$, and graphically by:

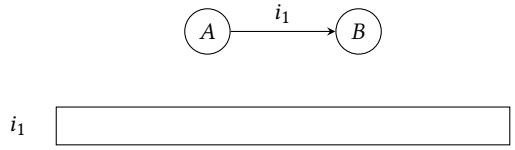

Fig. 9. Q6

PostgreSQL evaluates the query by performing a bitmap heap scan.

Q7:

Cypher:

```
match (n1)-[e1]->(n2)-[e2]->(n3)-[e3]->(n4),
(n3)-[e4]->(n5), (n3)-[e5]->(n6), (n3)-[e6]->(n7)
where e2.time_start = e1.time_start and e3.time_start =
e1.time_start and e4.time_start = e1.time_start and
e5.time_start = e1.time_start and e6.time_start =
e1.time_start and e2.time_end = e1.time_end and
e3.time_end = e1.time_end and e4.time_end = e1.time_end
and e5.time_end = e1.time_end and e6.time_end =
e1.time_end return count(*);
```

SQL:

```
select * from binned_intervals_interactions as e_1 left
join binned_intervals_interactions as e_2 on e_2.source =
e_1.target and e_1.tstart = e_2.tstart and e_1.tend =
e_2.tend left join binned_intervals_interactions as e_3
on e_3.source = e_2.target and e_1.tstart = e_3.tstart
and e_1.tend = e_3.tend left join
binned_intervals_interactions as e_4 on e_4.source =
e_2.target and e_1.tstart = e_4.tstart and e_1.tend =
e_4.tend left join binned_intervals_interactions as
e_5 on e_5.source = e_2.target and e_1.tstart =
e_5.tstart and e_1.tend = e_5.tend left join
binned_intervals_interactions as e_6 on e_6.source =
e_2.target and e_1.tstart = e_6.tstart and e_1.tend =
e_6.tend where e_3.interaction_id < e_4.interaction_id
and e_4.interaction_id < e_5.interaction_id and
e_5.interaction_id < e_6.interaction_id;
```

And finally, Query 7 can be represented by the predicates $P_{top} = target(edge_1) = source(edge_2) \wedge target(edge_2) = source(edge_3) \wedge target(edge_3) = source(edge_4) \wedge target(edge_3) = source(edge_5) \wedge target(edge_3) = source(edge_6)$ and $P_{temp} = tstart(edge_1) = tstart(edge_2) = tstart(edge_3) = tstart(edge_4) = tstart(edge_5) = tstart(edge_6) \wedge target(edge_1) = target(edge_2) = target(edge_3) = target(edge_4) = target(edge_5) = target(edge_6)$.

Query 7 can be represented graphically by:

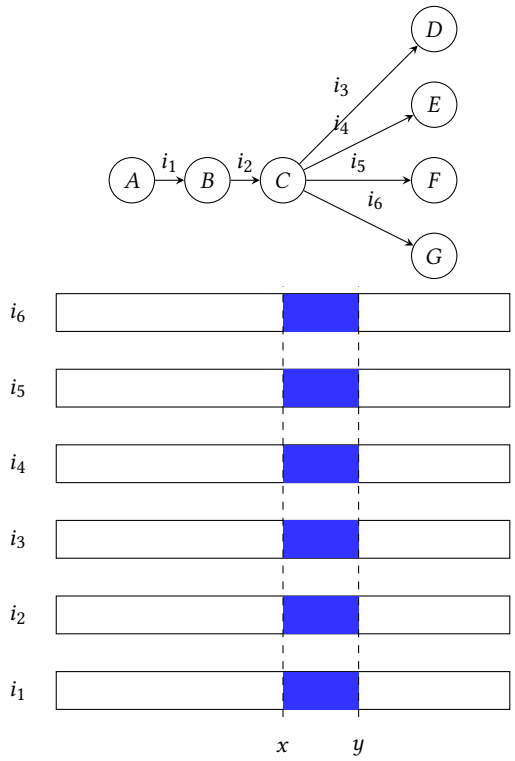

Fig. 10. Q7. The time windows of interactions $i_2, \ldots, i_6$ should be exactly equal to $[x, y]$, defined by the time window of the first interaction i_1.

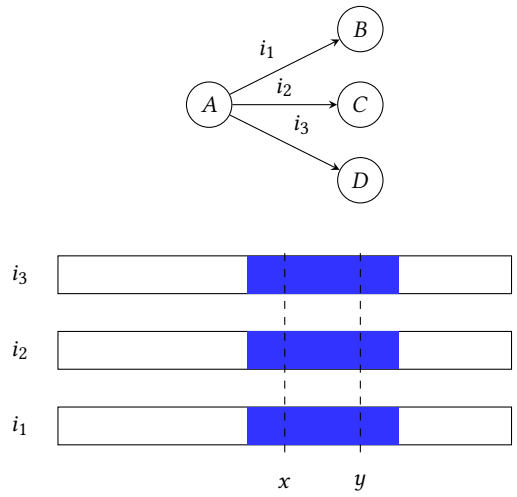

Fig. 11. Q1a.

Q1a:
Cypher:

```
match (n1)-[e1]->(n2), (n1)-[e2]->(n3),
(n1)-[e3]->(n4), e1.time_start >= 398144000 and
e1.time_end <= 401772800 and e2.time_start <= 398144000
and e2.time_end >= 401772800 and e3.time_start <=
398144000 and e3.time_end >= 401772800 return count(*);
```

SQL:

```
select * from interactions as e_1
left join interactions as e_2 on e_1.source = e_2.source
left join interactions as e_3 on e_1.source = e_3.source
where
e_1.tstart <= 398144000 and e_1.tend >= 401772800 and
e_2.tstart <= 398144000 and e_2.tend >= 401772800 and
e_3.tstart <= 398144000 and e_3.tend >= 401772800 and
e_1.interaction_id < e_2.interaction_id and e_2.interaction_id < e_3.interaction_id;
```

Query 1a can be graphically represented by

Q2a:
Cypher:

```
match (n1)-[e1]->(n2)-[e2]->(n3)-[e3]->(n4)-[e4]->
(n5)-[e5]->(n6) where e1.time_start >= 400000000
and e1.time_end <= 420000000 and e2.time_start >=
400000000 and e2.time_end <= 420000000 and e3.time_start
>= 400000000 and e3.time_end <= 420000000 and
e4.time_start >= 400000000 and e4.time_end <=
420000000 and e5.time_start >= 400000000 and e5.time_end
<= 420000000 return count(*);
```

SQL:

```
select * from interactions as e_1
left join interactions as e_2 on e_2.source = e_1.target
left join interactions as e_3 on e_3.source = e_2.target
left join interactions as e_4 on e_4.source = e_3.target
left join interactions as e_5 on e_5.source = e_4.target
where e_1.tstart >= 400000000 and e_1.tend <= 420000000 and
e_2.tstart >= 400000000 and e_2.tend <= 420000000 and
e_3.tstart >= 400000000 and e_3.tend <= 420000000 and
e_4.tstart >= 400000000 and e_4.tend <= 420000000 and
e_5.tstart >= 400000000 and e_5.tend <= 420000000
```

Query 2a can be graphically represented by

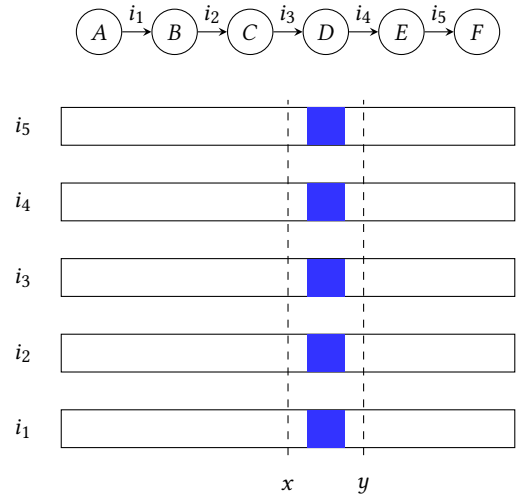

Fig. 12. Q2a

Q3a:
Cypher:

```
match (n1)-[e1]->(n2)-[e2]->(n3)-[e3]->(n4)-[e4]->
(n5)-[e5]->(n6) where e1.time_start <= e2.time_end and
e1.time_start <= e3.time_end and e2.time_start <=
e3.time_end and e1.time_start <= e4.time_end and
e2.time_start <= e4.time_end and e3.time_start <=
e4.time_end and e1.time_start <= e5.time_end and
e2.time_start <= e5.time_end and e3.time_start <=
e5.time_end and e4.time_start <= e5.time_end
return count(*);
```

SQL:

```
select * from interactions as e_1
left join interactions as e_2 on e_2.source = e_1.target
left join interactions as e_3 on e_3.source = e_2.target
left join interactions as e_4 on e_4.source = e_3.target
left join interactions as e_5 on e_5.source = e_4.target
where e_1.tstart < e_2.tend and
e_1.tstart < e_3.tend and
e_2.tstart < e_3.tend and
e_1.tstart < e_4.tend and
e_2.tstart < e_4.tend and
e_3.tstart < e_4.tend and
e_1.tstart < e_5.tend and
e_2.tstart < e_5.tend and
e_3.tstart < e_5.tend and
e_4.tstart < e_5.tend
```

Q3a can be graphically represented by

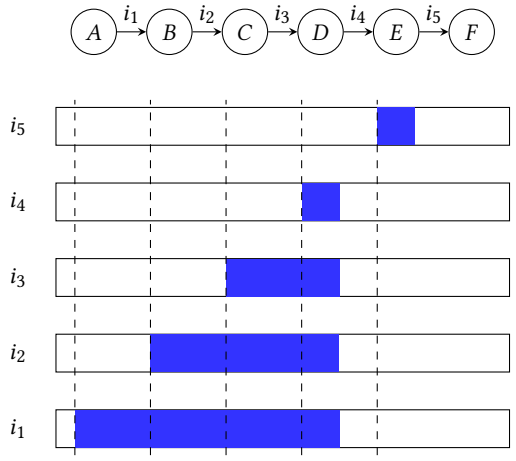

Fig. 13. Q3a.

Q4a:
Cypher:

```
match (n1)-[e1]->(n2)-[e2]->(n3)-[e3]->(n4) where
e1.time_start <= e2.time_start and e2.time_start
<= e3.time_start return count(*);
```

SQL:

```
select * from interactions as e_1
left join interactions as e_2 on e_2.source = e_1.target
left join interactions as e_3 on e_3.source = e_2.target
where e_1.tstart < e_2.tstart and
e_2.tstart < e_3.tstart;
```

Q4a can be represented by

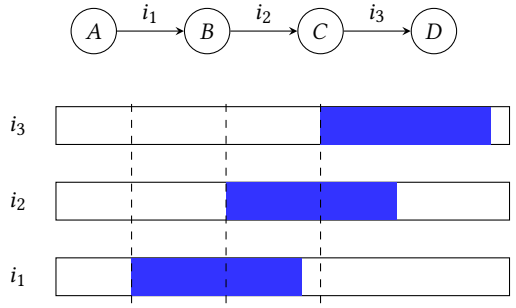

Fig. 14. Q4a.

PostgreSQL evaluates the query by a nested loop operation.