Bitemporal Property Graphs to Organize Evolving Systems

Towards the development of a graph model, database, and query language to represent, store, and query temporal graphs

Christopher Rost[1,2][0000−0003−4217−9312], Philip Fritzsche[1,4], Lucas Schons[1,2], Maximilian Zimmer[1], Dieter Gawlick[3][0000−0002−7882−0565], and Erhard Rahm[1,2][0000−0002−2665−1114]

[1] University of Leipzig, Augustusplatz 10, 04109 Leipzig, Germany
[2] ScaDS.AI Dresden/Leipzig, Humboldtstraße 25, 04105 Leipzig, Germany
{rost,rahm}@informatik.uni-leipzig.de
[3] Oracle Server Technologies, 500 Oracle Parkway, CA 94065, USA
dieter.gawlick@oracle.com
[4] Oracle Labs Zurich, Hardstrasse 201, 8005 Zürich, Switzerland
philip.fritzsche@oracle.com

Abstract. This work is a summarized view on the results of a one-year cooperation between Oracle Corp. and the University of Leipzig. The goal was to research the organization of relationships within multi-dimensional time-series data, such as sensor data from the IoT area. We showed in this project that temporal property graphs with some extensions are a prime candidate for this organizational task that combines the strengths of both data models (graph and time-series). The outcome of the cooperation includes four achievements: (1) a bitemporal property graph model, (2) a temporal graph query language, (3) a conception of continuous event detection, and (4) a prototype of a bitemporal graph database that supports the model, language and event detection.

Keywords: Graph database · Temporal graph query language · Temporal property graph model.

1 Introduction

The main goal of the project was to research the suitability of temporal property graphs for the organization of multi-dimensional time series data, such as sensor data from the IoT area. Even though this data has a very high information content in itself, the used data structure offers no possibility of depicting or describing the relationships between entities, e. g., those producing time-series.

For example, an aircraft, like the one in Fig. 1, is a complex system made up of a large number of bigger and smaller components, many of which are equipped with a wide variety of sensors that continuously capture data to determine its

current status. One component (also denoted as an asset) is often also a complex system of smaller components, e.g., the airplane turbine, most of them again equipped with sensors.

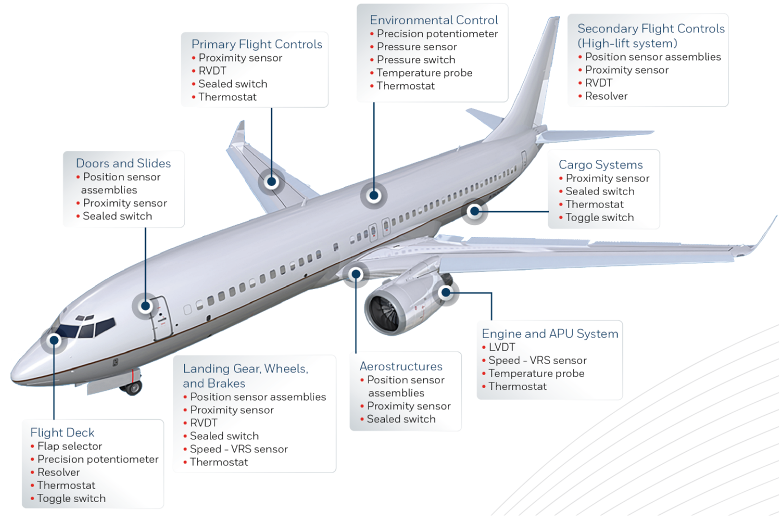

Fig. 1. Sensors of an aeroplane [12].

Imagine that each sensor in such a complex system delivers a time series of values, e.g. temperatures, rotational speeds, centrifugal forces, accelerations, etc. To model the relationships between the sensors, the components, and the sensors, as well as the components between each other, a data structure is required that can model such a complex, heterogeneous network and allow structural and content changes to the network and store exactly when these changes took place. Having such a network (or graph), one can query for these relationships and find correlations that might be hidden by looking at the time series isolated from each other.

We figured out in this project that temporal property graphs with temporal extensions are the prime candidate for this organizational task that combines strengths of both data models (graph and time series). Several requirements emerged for this technology combination, including (1) the need for a rich graph data model with full auditing of the graph's evolution - in the observed real world as well as in the graph database, (2) the need for a declarative query language to match patterns in a changing graph and (3) the need for an event detection mechanism through a continuous evaluation of registered patterns against graph changes.

Within the project lifetime, we achieved the following four contributions:

1. TPGM$^+$: A bitemporal property graph model supporting property changes

2. T-PGQL: A temporal graph query language
3. CGN: Event detection through Continuous Graph Notifications
4. BiTeGra: A bitemporal graph database based on RDBMS

After a brief overview of related work (Section 2), we will begin by introducing a new bitemporal property graph model which supports changing properties in Section 3. We further give an introduction to T-PGQL, a temporal graph query language based on Oracle's PGQL, in Section 4. The Continuous Graph Notification (CGN), a technology to continuously evaluate registered T-PGQL queries for event detection purposes, is introduced in Section 5. In section 6, a prototype of a temporal graph database based on a relational database is presented, which supports the developed graph data model, the query language and event detection engine. We sum up the work and give an outlook to ongoing work in Section 7.

2 Related work

A prominent graph model is the Property Graph Model [26,2], which defines a directed graph where vertices and edges can have an arbitrary number of *properties* that are typically represented as key-value pairs. Since this model has no native maintenance of a graph's evolution, several other data models for temporal graphs exist in the literature [11,18,5,32], including duration-based graphs [33], interval-based graphs [28,27,4] and snapshot graphs [17]. The models differ also in other aspects, e.g., supported time-dimensions or possible updates, so that there is not yet a consensus about the most promising approach [27].

Well-known declarative query languages for property graphs are: Cypher [21,9], mainly supported by the graph database engine Neo4j, Gremlin [25], part of the Apache TinkerPop graph computing framework, and PGQL [24] which is implemented in Oracle Spatial and Graph [6] and PGX [31]. These languages support date and time formats for vertex and edge properties or offer a type for expressing a duration, but they are based on the static property graph model and therefore do not support temporal property graphs, where the time is a dimension of the data model. Although temporal language extensions were introduced for relational databases a decade ago [15,19,14], for temporal graphs, there are just a few concepts of query languages that support the additional time dimension(s) and allow querying past graph states or formulate path patterns with chronological order. Temporal-GDL [27] and T-GQL [7] are two examples of temporal graph query languages that were recently introduced in the literature.

According to the *continuous* querying of graph data, most concepts are based on the streaming model, i. e., relationships (and in some models entities) are represented as an event stream that can be analyzed in (near) real-time. Frameworks supporting streaming graphs maintain a dynamically changing graph dataset under a series of updates and queries to the graph data [3]. In contrast to native graph stream frameworks, like STINGER [8], LLAMA [20], GraphIn [30] and GraphTau [13], the system Graphflow [16] is a in-memory property graph storage that supports continuous subgraph queries.

3 TPGM$^+$: A bitemporal property graph model

One of the most famous and important graph models is the Property Graph Model (PGM) [26,2], which was introduced in 2010. A graph of this model is characterized by vertices (or nodes) and directed edges, e.g., users in a social network and their friendship relationships. The graph is labeled, which means, according to the definition, nodes and edges can be assigned one or more type labels, e.g. *User* or *Friendship*. Nodes and edges can have zero or more so-called properties, which are represented as key-value pairs and describe the entity or relationship in more detail, e.g., *name:Christopher* or *city:Leipzig*. By default, a property graph is also a multigraph, i.e., it is permitted to have multiple edges (also called parallel edges) between two nodes.

Based on Property Graph Model (PGM), we previously introduced the Temporal Property Graph Model (TPGM) [28] and formally defined it in [27]. It extends the PGM mainly by adding two additional attributes to each vertex and edge that describe its validity or lifetime in a bitemporal way. One attribute defines the valid-time (also called application time) which describes the validity of the entity or relationship in the real world, e.g., when a call between two users starts and ends. The other attribute defines the transaction-time (also called system-time) which describes when the information about the existence of the entity or relationship was inserted into the database. This bitemporal modeling is already used in relational databases.

One downside of the TPGM is the weak support for property changes. If a property of a vertex or edge changes, i.e., it is created updated or removed, a logical copy of this element is created that stores the updated state and the information about the time of the update. All properties that are not affected by the changes stay unchanged and exist as overhead. High frequent changes of properties and their values thus result in a huge amount of duplicated elements.

To overcome this, we created in this collaboration the $TPGM^+$, an extended version of the TPGM that adds bitemporal modeling to the layer of properties. For each $TPGM^+$ graph, two linerary ordered discrete time domains Ω exist: Ω^{tx} for the transaction time and Ω^{val} for the valid-time. Each instant in time is a time point t_i. The linear ordering is described as $t_i < t_{i+1}$, i.e., t_i happened before t_{i+1}. Per time domain, each vertex edge **and property value** has an associated time-period τ. Given $t_s, t_e \in \Omega$, a time-period $\tau = [t_s, t_e)$ is defined as a close-open time interval starting at and including t_s and ending at but excluding t_e. The time points of τ are thus a set $\{t \mid t \in \Omega \wedge t_s \leq t < t_e\}$. The length or size of a period τ is defined by $l(\tau)$. Default values for lower and upper bounds are $t_{min} = -\infty$ and $t_{max} = \infty$.

Further, we defined several integrity constraints of a $TPGM^+$ graph, to ensure the consistency of the graph at each point in time. Each constraint is valid per time domain.

1. *Unique vertices, edges and properties* At every point in time, vertices, edges and properties are unique, i.e., exist at most once. The uniqueness constraint of vertices and edges is a combined key of the element's identifier and the end-timestamp. The uniqueness constraint of a property is its name.

2. *Referential integrity of edges* For each edge, the time intervals associated with its source and target vertices must contain the edge's time interval. In other words, an edge can only exist when its incident vertices exist.
3. *Referential integrity of properties* For a property value, the interval of the vertex/edge must contain the interval of the property value. In other words, a property value can only exist when the corresponding vertex/edge exists.
4. *Constant edges* Source and target vertices of an edge never change at existence.
5. *Constant types* Vertex and edge labels, as well as property key names, never change.

4 T-PGQL: A temporal graph query language

We will now have a look at some preliminaries needed to understand the extensions we made to the graph query language PGQL, that are explained afterward.

4.1 Preliminary: PGQL - A (Non-temporal) Graph Query Language

PGQL [24,23] combines graph pattern matching with SQL-like syntax and functionality and has full-blown support for regular path queries and graph construction. Because its syntax is SQL-like, the language is intuitive to use for existing SQL users. Furthermore, PGQL queries return a “resultset” with variables and bindings, just like in SQL.

Matching graph patterns is one main functionality of PGQL realized by a SELECT-Query. Similar to SQL, a SELECT query is composed of several clauses, starting with the mandatory SELECT clause and FROM-Clause. In PGQL, the SELECT clause defines the returned result entities of the query. Since the result of a SELECT query is always a table, the SELECT clause defines the attributes of the result table. The following syntactic structure of a PGQL SELECT query and clause is taken from [23]:

```
SelectQuery    ::=  SelectClause
                    FromClause
                    WhereClause?
                    ...
SelectClause    ::= 'SELECT' 'DISTINCT'? ExpAsVar ( ',' ExpAsVar )*
                    | 'SELECT' '*'
ExpAsVar    ::= ValueExpression ( 'AS' VariableName )?
```

The graph pattern to be matched is defined by the FROM clause that includes one or multiple MATCH clauses. A MATCH clause defines a path- or graph pattern, where a graph pattern is a composition of path patterns.

The following syntactical definitions of the FROM- and MATCH clauses are taken from [23].

```
FromClause      ::= 'FROM' MatchClause ( ',' MatchClause )*
MatchClause     ::= 'MATCH' ( PathPattern | GraphPattern ) OnClause?
GraphPattern    ::= '(' PathPattern ( ',' PathPattern )* ')'
PathPattern     ::=   SimplePathPattern | ShortestPathPattern | ...
```

Path patterns describe topology constraints, where a topology constraint is a composition of one or multiple vertices and edges. A vertex or edge is optionally identified by a variable, i.e., a symbolic name to reference it in other clauses. It is also possible to define one or more label predicates directly in the pattern. A PGQL SELECT query returning all person and movie names that match this pattern can be formulated as follows:

```
1 SELECT p.name, m.movie
2 FROM MATCH (p:Person)-[l:like|dislike]->(m:Movie)
```

For further information, we refer to the official documentation of PGQL, available at [23].

4.2 Extensions of PGQL to Query TPGM$^+$ Graphs

In current graph query languages patterns (or paths) are searched in the whole available graph database without observance of any evolution. Having a temporal graph modeled by the TPGM$^+$ (see Sect. 3), several new requirements arise for a query language to support temporal features of the model. In this work, we limit to the retrieval of data. The manipulation of data as well as functions for the definition and manipulation of database structures are not considered yet.

Access of Temporal Attributes Our data model tracks changes in a bitemporal model on a level of vertices, edges and properties. We introduce the possibility to add these attributes via projection to the resulting relation and to use these attributes in expressions for selection.

We distinguish four different temporal characteristics of a vertex, edge and property: the period itself, its lower bound timestamp (inclusive), its upper bound timestamp (exclusive) and length/duration of a period.

The identifier for the period is `VAL_TIME` for the valid time domain and `TX_TIME` for the transaction time domain. A period is, like in SQL, not a data type but a type definition [19]. The textual representation a period projection is a concatenated result of both period bounds in the form of: `[{from},{to})`. A period type can be used for several predicates, as later described. The identifiers for the period bounds are `VAL_FROM` and `VAL_TO` for the valid time domain and `TX_FROM` and `TX_TO` for the transaction time domain. The result of a period-bound access is a timestamp.

To access these attributes of a vertex or edge, they can be used like a property access by dot notation, e.g., `var1.TX_FROM`, `var2.VAL_TIME`. According to the temporal attributes of a property, the same suffix can be used on a property access expression, like `var1.prop1.TX_FROM` or `var2.prop2.VAL_TIME`. The following syntax describes the notation.

```
TimeIdentifier  ::= 'TX_TIME' | 'VAL_TIME' | 'TX_FROM' | 'TX_TO'
                  | 'VAL_FROM' | 'VAL_TO'
Property        ::= Identifier
VarRef          ::= Identifier
PropRef         ::= VarRef '.' Property
ElementTimeRef  ::= VarRef '.' TimeIdentifier
PropertyTimeRef ::= PropRef '.' TimeIdentifier
TimeRef         ::= ElementTimeRef | PropertyTimeRef
```

For example, the following query returns all available temporal characteristics of person vertices and their name property.

```
1 SELECT n.TX_FROM, n.TX_TO, n.TX_TIME,
2        n.VAL_FROM, n.VAL_TO, n.VAL_TIME,
3        n.name.TX_FROM, n.name.TX_TO, n.name.TX_TIME,
4        n.name.VAL_FROM, n.name.VAL_TO, n.name.VAL_TIME
5 FROM MATCH (n:Person) ON student_network
```

Besides the period of a graph element or its property, the length/duration of this period can be queried. We introduce a `LENGTH([unit,]period)` expression which consumes a period access identifier (`period`) as argument and an optional unit (i.e., `YEAR, QUARTER, MONTH, WEEK, DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). If no unit is given, the default unit `MILLISECOND` is used. This expression returns a numerical value that can be used within several expressions, e.g., binary constraints. The following query returns the duration in days of the valid time period of a person's name property for all persons whose valid time exceeds one day.

```
1 SELECT LENGTH(DAY, n.name.VAL_TIME)
2 FROM MATCH (n:Person) ON student_network
3 WHERE LENGTH(DAY, n.name.VAL_TIME) > 1
```

Temporal Filtering and Chronological Pattern Matching Graphs with a managed valid-time are intended for meeting the requirements of applications that are interested in capturing time periods during which the data is (believed to be) valid in the real world. For each vertex or edge, as well as their properties, a valid-time period is available. As described in **??**, the identifier of this period is `VALID_TIME` and the beginning and ending bounds are `VAL_FROM` and `VAL_TO`. They can be used as a suffix to variables and property access. Analogous to the transaction time attributes, the valid time period returns a new *period* type definition and the bounds return a timestamp type. Latter can be used like regular timestamp attributes, e.g., within binary relations in the WHERE clause. The WHERE clause is defined as follows [23]. Note that `TemporalExpression` is a temporal extension of our work.

```
WhereClause         ::= 'WHERE' ValueExpression
```

```
ValueExpression     ::= VariableReference
                        | PropertyAccess
                        | ...
                        | TemporalExpression
TemporalExpression  ::= ElementTimeRef
                        | PropertyTimeRef
                        | Overlaps
                        | Equals
                        | Precedes
                        | Succeeds
                        | Contains
Overlaps            ::= TimeRef 'OVERLAPS' TimeRef
Equals              ::= TimeRef 'EQUALS' TimeRef
Precedes            ::= TimeRef ('IMMEDIATELY')? 'PRECEDES' TimeRef
Succeeds            ::= TimeRef ('IMMEDIATELY')? 'SUCCEEDS' TimeRef
Contains            ::= TimeRef 'CONTAINS' TimeRef
```

For example, to retrieve all students that studied in a University in Leipzig as of February 15, 2019, one can express the query by accessing the period bounds in predicates of the WHERE clause:

```
1 SELECT n.name
2 FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
3 WHERE u.city = 'Leipzig'
4     AND s.VAL_FROM <= DATE '2019-02-15'
5     AND s.VAL_TO > DATE '2019-02-15'
```

To simplify the expression of such predicates, several language extensions are further defined. For example, we can use one of the period predicates provided in SQL:2011 for expressing conditions involving periods: `CONTAINS`, `OVERLAPS`, `EQUALS`, `PRECEDES`, `SUCCEEDS`, `IMMEDIATELY PRECEDES`, and `IMMEDIATELY SUCCEEDS`. All period predicates need two expressions that return a period as arguments, except for `CONTAINS`, which also allows a timestamp as a second argument. The query above can be simplified by using the `CONTAINS` predicate.

```
1 SELECT n.name
2 FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
3 WHERE u.city = 'Leipzig'
4     AND s.VALID_TIME CONTAINS DATE '2019-02-15'
```

To retrieve all students of Universities of Leipzig who are matriculated from January 1, 2018, to January 1, 2019, one could formulate the query by using a temporal condition, in our case, the `OVERLAPS` predicate:

```
1 SELECT n.name
2 FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
3 WHERE u.city = 'Leipzig'
```

```
4      AND s.VALID_TIME OVERLAPS PERIOD(DATE '2018-01-01',
5                                       DATE '2019-01-01')
```

In the example, we also show a new period constructor expression that allows the creation of a period instance from two timestamps t_1 and t_2 with $t_1 <= t_2$. The timestamps can be defined by any expression that returns a date or timestamp instance. For example, they can be created through a date or timestamp constructor as in the example or extracted from a graph element or property through a period bound identifier, e.g., `PERIOD(DATE '2018-01-01', x.VAL_TO)`.

Also, the transaction time period of elements and properties can be used in predicates of this kind. The following query returns the name and system-time period for students matriculated in a University located in Leipzig where the information about the matriculation was added to the database after March 1, 2018. The syntax part `FOR TX_TIME ALL` is explained in the next section.

```
1 SELECT n.name, n.TX_TIME
2 FROM MATCH (n:Person)-[s:studiedAt]->(u:University) FOR TX_TIME ALL
3 WHERE u.city = 'Leipzig'
4     AND s.TX_FROM >= TIMESTAMP '2018-03-01 00:00:00'
```

The classical graph pattern matching is used to find a matching subgraph in the graph database that matches exactly with the defined query pattern. In the well-known static scenario, a query pattern has no information about the chronological ordering of the given entities and relationships. For example, a pattern like `(p1:Person)-[l1:likes]->(p2:Person)-[l2:likes]->(p3:Person)` includes no information about when the likes happened or if one like happened before the other of if they happened at the same time. To overcome this lack, the above introduced temporal predicates can be used to enrich such patterns with temporal information. The following code exemplifies that:

```
1 SELECT p1.name, p2.name, p3.name
2 FROM MATCH (p1:Person)-[l1:likes]->(p2:Person)-[l2:likes]->(p3:Person)
3 WHERE l1.VAL_TIME PRECEDES l2.VAL_TIME
```

Here we add a constraint, that the like between p1 and p2 must happen before the like between p2 and p3. These kinds of predicates can thus be used to define a temporal ordering in a path pattern.

Graph Snapshot Retrieval and Historical Pattern Matching T-PGQL can be used to find matching subgraphs in a specific state of the temporal graph with respect to the transaction time domain. This state can be a snapshot at a defined timestamp or all changes of a given period. Former represents a valid snapshot graph without multiple versions of a single instance. The transaction time dimension is reduced to a single point in time. Latter is again a temporal property graph that can have multiple versions of a single instance. The transaction time dimension is reduced to a range in time.

To search for a defined pattern using this kind of time traveling, we extended PGQL's `FROM` clause with a clause similar to the SQL extension for temporal tables. The name of the transaction time period definition is fixed to `TX_TIME`. To query the historical data, the clause `FOR TX_TIME {predicate}` has to be used directly after a match clause (see Section 4.1). To define the timestamp or period to query for, we provide four predicates as syntactic extensions:

```
GraphMatch          ::= 'MATCH' PathPattern OnClause? SysTimeCond
SysTimeCond         ::= 'FOR' 'TX_TIME' ( AsOf | FromTo
                                        | BetweenAnd | 'ALL')
AsOf                ::= 'AS' 'OF' TimeRef
FromTo              ::= 'FROM' TimeRef 'TO' TimeRef
BetweenAnd          ::= 'BETWEEN' TimeRef 'AND' TimeRef
```

The argument `TimeRef` could be any expression returning a timestamp attribute, i.e., a date or timestamp constructor (e.g., `TIMESTAMP('2020-01-01')`), current timestamp expression (`CURRENT_TIMESTAMP`) or access expressions of temporal attributes for vertices, edges or properties. If the `FOR TX_TIME` clause is not used, the result will show the current data, as if one had specified `FOR TX_TIME AS OF CURRENT_TIMESTAMP`. Thus, the following queries are equal:

```
1 SELECT n.name
2 FROM MATCH (n:Person)
3     ON student_network
```

```
1 SELECT n.name
2 FROM MATCH (n:Person)
3     ON student_network FOR TX_TIME AS OF CURRENT_TIMESTAMP
```

The first predicate `AS OF {timestamp}` is used to see the graph as it was at a specific point in time in the presence or past. The following example query retrieves the graph as it was on 1st February 2020 at 1 pm.

```
1 SELECT n.name
2 FROM MATCH (n:Person) FOR TX_TIME AS OF TIMESTAMP '2020-02-01 13:00'
```

The next query using the `BETWEEN {timestamp} AND{timestamp}` predicate will show all graph elements that were visible at any point between two specified points in time. It works inclusively, i.e., an element visible exactly at the start or exactly at the end will be added to the result set too.

```
1 SELECT n.name
2 FROM MATCH (n:Person) FOR TX_TIME BETWEEN TIMESTAMP '2020-02-01 12:00'
3     AND TIMESTAMP '2020-02-28 12:00:00'
4
```

The extension `FROM {timestamp} TO {timestamp}` will also show all elements that were visible at any point between two specified points in time, including start, but excluding end.

```
1 SELECT n.name
2 FROM MATCH (n:Person) FOR TX_TIME FROM TIMESTAMP '2020-02-01 12:00'
3     TO TIMESTAMP '2020-02-28 12:00:00'
```

To query for the current state and complete history of a given pattern the predicate `ALL` can be used.

```
1 SELECT n.name
2 FROM MATCH (n:Person) FOR TX_TIME ALL
```

In PGQL, it is possible to define multiple patterns within a single `FROM` clause by using multiple match clauses separated by a comma. Our extension can be used within each of these match expressions. This provides a flexible mechanism to define patterns with parts occurring at different times. For example, to find people who currently liked a post that already existed on January 1st, 2020, the following query can be used:

```
1 SELECT m.firstName, m.lastName
2 FROM MATCH (p:Post) FOR TX_TIME AS OF DATE '2020-01-01',
3      MATCH (m:Person)-[:likes]->(p) // current
```

Another query scenario is to find a sensor that is currently (`FOR TX_TIME AS OF CURRENT_TIMESTAMP`) connected to an asset that existed in the past (`FOR TX_TIME AS OF DATE '...'`). Besides a path pattern, a graph pattern is a concatenated list of path patterns in round brackets, which can be also specified after the match keyword. If a transaction time predicate should be applied to a set of path patterns, it can be used after such a graph pattern, as can be seen in the following example.

```
1 SELECT m.firstName, m.lastName
2 FROM MATCH (
3     (p:Post)-[:hasTag]->(t:Tag)-[:inClass]->(tc:TagClass),
4     (m:Person)-[:likes]->(p:Post)
5     ) FOR TX_TIME AS OF DATE '2020-01-01'
```

Bitemporal Queries A TPGM graph has both a managed transaction- and valid-time domain. Graph elements, as well as their properties, are associated with both transaction-time and valid-time periods. This concept is very useful for use cases where it is necessary to capture both the periods during which facts were believed to be true in real-world as well as periods during which those facts were recorded in the database.

For example, a student changes his address. Typically the address changes legally at a specific time, but it is not changed in the database concurrently with the legal change. In that case, the transaction-time period automatically records when the new address is known to the database, and the valid-time period records when the address was legally effective. Successive updates to bitemporal graphs

can journal complex twists and turns in the state of knowledge captured by the database [19].

Queries on bitemporal graphs can specify predicates on both dimensions to qualify rows that will be returned as the query result. For example, the following query returns all students of Universities in Leipzig who are matriculated as of December 1, 2019, recorded in the graph database as of January 1, 2020.

```
SELECT n.name
FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
    ON student_network FOR TX_TIME AS OF DATE '2020-01-01'
WHERE u.city = 'Leipzig'
    AND s.VALID_TIME CONTAINS DATE '2019-12-01'
```

Query the Evolution of a Property Every vertex and edge can have zero, one or more properties in form of key/value pairs, where the key represents the name of the property. For every property, transaction-time versioning is supported to track the addition of new properties, changes in values or deletion of exiting properties. Thus a property behaves like a vertex or an edge. By inserting a vertex or edge with properties into the database, each gets the same system-time period as the respective vertex or edge.

The previous introduced FROM clause extensions (see section 4.2) have also an effect on the property retrieval since their transaction-time period will be considered, too.

In addition, each property contains a valid-time period. For example, each `University` vertex has a property `studentCount` whose value is periodically updated. Each value has an application time period that defines, for which time the value was true. If no further condition is specified for an application time enabled property, all values are returned.

```
SELECT u.name as name, u.studentCount as cnt
FROM MATCH (u:University)
WHERE u.city = 'Leipzig'
```

Result:

```
+---------------------+-------+
|        name         |  cnt  |
+---------------------+-------+
| Leipzig University  | 28004 |
| Leipzig University  | 28797 |
| Leipzig University  | 29061 |
+---------------------+-------+
```

To retrieve the information of the validity of the values, the period of validity can be selected.

```
1 SELECT u.name as name, u.studentCount as cnt,
2     u.studentCount.VALID_TIME as validity
3 FROM MATCH (u:University)
4 WHERE u.city = 'Leipzig'
```

Result:

```
+--------------------+-------+--------------------------------------+
|        name        |  cnt  |               validity               |
+--------------------+-------+--------------------------------------+
| Leipzig University | 28004 | [2016-04-01 00:00, 2017-04-01 00:00) |
| Leipzig University | 28797 | [2017-04-01 00:00, 2018-04-01 00:00) |
| Leipzig University | 29061 | [2018-04-01 00:00, 2019-04-01 00:00) |
+--------------------+-------+--------------------------------------+
```

All previously introduced conditions of the `WHERE` clause that are applicable for vertices and edges can be used by properties too.

```
1 SELECT u.name, u.studentCount, u.studentCount.VALID_TIME as validity
2 FROM MATCH (u:University)
3 WHERE u.city = 'Leipzig'
4     AND u.studentCount.VALID_TIME CONTAINS TIMESTAMP '2018-01-01 00:00'
```

Result:

```
+--------------------+-------+--------------------------------------+
|        name        |  cnt  |               validity               |
+--------------------+-------+--------------------------------------+
| Leipzig University | 28797 | [2017-04-01 00:00, 2018-04-01 00:00) |
+--------------------+-------+--------------------------------------+
```

4.3 Aggregations

An aggregate function allows performing a calculation on a set of values to return a single scalar value. Aggregate functions are used with the GROUP BY and HAVING clauses of the query.

In this work, the PGQL language was extended by two functions: `FIRST(date or timestamp values)` and `LAST(date or timestamp values)`. The former returns the chronological earliest date or timestamp, while the latter returns the chronological last.

The following query returns the first beginning of a *studentAt* relationship according to the application-time and system-time domain.

```
1 SELECT FIRST(s.VAL_FROM) as earliestStart,
2        FIRST(s.startTT) as earliestTx
3 FROM MATCH ()-[s:studiedAt]->()
```

Result:

```
+---------------------+---------------------+
|    earliestStart    |     earliestTx      |
+---------------------+---------------------+
| 1409-04-01 00:00:00 | 2006-05-12 14:45:22 |
+---------------------+---------------------+
```

The following query can be used to answer the question: For universities of a certain city, when was the first time a student began his studies, and when was the most recent time?

```
1 SELECT u.city,
2        FIRST(s.VAL_FROM) as earliestStart,
3        LAST(s.VAL_FROM) as latestStart
4 FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
5 GROUP BY u.city
```

Result:

```
+---------+---------------------+---------------------+
|  city   |    earliestStart    |     latestStart     |
+---------+---------------------+---------------------+
| Leipzig | 1409-04-01 00:00:00 | 2020-04-01 00:00:00 |
| Berlin  | 1810-04-01 00:00:00 | 2020-04-01 00:00:00 |
| Munich  | 1472-04-01 00:00:00 | 2020-04-01 00:00:00 |
+---------+---------------------+---------------------+
```

5 CGN: Event detection through Continuous Graph Notifications

We introduced T-PGQL as a query language for executing SELECT queries on TPGM$^+$ graphs in the previous Section 4. Querying a graph with T-PGQL is usually done in a single graph query execution, i.e., a user formulates a SELECT query, executes that query on the current state of a database system that maintains a TPGM$^+$ graph and gets a result in form of a table back. In this way, one can query for current and historical data of the graph where one query leads to one fixed result. Subsequent changes of the graph are not taken into account unless the query is executed again which leads to a result that recognizes all transactions that are made until the time when the query was executed.

Talking about changes in a graph leads to the observation of events. An event itself can be everything that happened at a defined point in time, e. g., an asset's sensor captures a temperature or a user liked a post in a social network. We semantically distinguish between *application-world events* and *transaction-world events*. Former is an event that happened in the observed real world. It can be described by a graph pattern and its predicates inside a graph query. In contrast, the discovery of an instance of a pattern in the data store at a specific time is a *transaction-world event*. I. e., the most recent commit (1) created an

instance of a pattern that did not exist before, e. g., a captured temperature of a sensor exceeds a threshold, or (2) destroys an instance of the pattern that already existed, e.g. a friendship relation between two users of a social network is removed.

The question arises, how a user can get a notification about a *transaction-world events*, i. e. when graph data changes that affect the elements that are accessed to create the query result. For RDBMS, there is a feature called Continuous Query Notification (CQN) [22] which is currently implemented by the Oracle database. It allows to register a SQL query and receive notifications when an event occurs that changes a table, i. e., that rows have been updated. This is useful in applications like near real-time monitoring, auditing applications, or for such purposes as mid-tier cache invalidation [22]. In this work, we adapt some concepts of the relational CQN to the graph domain. On a graph database that implements the TPGM^{+}, it is not only possible to simply execute a T-PGQL query, but also to register one. A registration is configured by the T-PGQL SELECT query itself, a registration validity period that specifies when the notification is enabled and when it will be disabled, a notification endpoint (e.g., a messaging queue) and a notification type. There are two types of graph query notifications: (1) the *Graph Change Notification (GCN)* and (2) the *Graph Query Result Change Notification (GRCN)* [34]. The GCN notifies the registrar if a transaction on the queried graph affects the graph elements queried. This does not automatically mean that the query result has also changed. To get a notification about an event that changes the graph in a way that affects the query result, the GRCN type must be used. Note that both types are a smart evaluation of a registered query, but not a query re-execution. This means the recipients are notified about the event but do not get the updated query result.

Assume the following T-PGQL query. The query describes the event of a temperature measurement above a value of 40 from a sensor that is part of an asset that is connected to an asset with id '42'. The projection of the query is the sensor value and its validity timestamp, which describes the time when the measure happened in the observed real world.

```
1 SELECT s.value, s.value.VAL_FROM
2 FROM MATCH (a1:Asset)-[p:partOf]->(a2:Asset)-[:hasSensor]->(s:Sensor)
3 WHERE a1.id = 42 AND s.type = 'temperature' AND s.value > 40
```

We assume that there are several matches for this pattern without recognizing the predicate of the value threshold, but none that fulfills this condition (i.e., all temperature values are below a value of 40). Further assume, that at a time t_1, the value of a sensor's property is updated from 39 to 40. If the query is registered by a GCN, a notification about that event is created, since one of the involved graph elements changed its state, but the query result is not affected. If the query is registered by a GRCN, no notification is created, since the query result does not change (it is still empty). Now, assume that at time t_2 the value of the sensor changes from 40 to 41. At this time, a notification is created for both types, since (1) a graph element that is part of the pattern changes in some

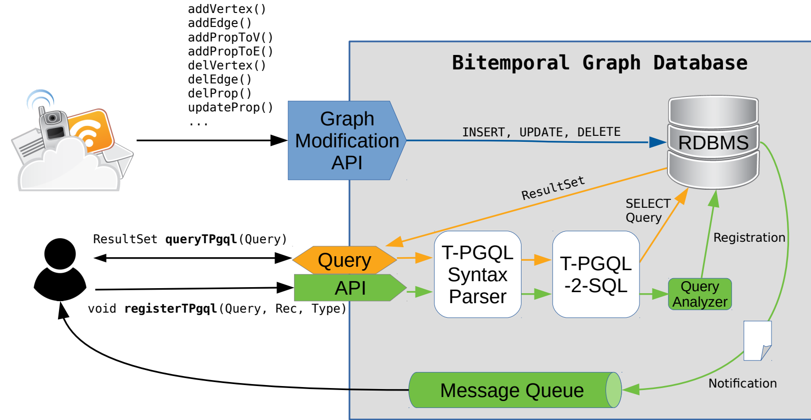

Fig. 2. Architectural draft of the bitemporal graph database

way and (2) the query result changes in the form that now one row is part of the result.

6 BiTeGra: A bitemporal graph management system

With the development of a bitemporal graph model, a query language for bitemporal graphs and the possibility of registering queries for continuous evaluation, the foundations for a graph database have been laid that unites all of the concepts considered. We will introduce an architectural draft of BiTeGra, a bitemporal graph database management system, and details of three main components including the graph to relation mapping, the graph modification features and the querying possibilities. Figure 2 shows an architectural draft of the BiTeGra system. The core of the system is a relational database management system with bitemporal table support. The graph data is stored in this database in a way that is described in Section 6.2.

6.1 Graph Modifications

A graph modification API provides an interface to manipulate graph data. There are methods to allow the following modifications to the maintained temporal graph:

- Add vertices/edges
- Add properties to vertices/edges
- Edit properties
- Edit valid-time attributes of vertices/edges
- Edit valid-time attributes of properties

– Delete properties from vertices/edges
– Delete vertices/edges

Each API call is internally translated to a SQL INSERT, UPDATE or DELETE query, depending on the used table schema. The API can be used by a wide variety of applications, e.g., the import of graph streams and other streaming and time-series data.

6.2 Schema mapping: Graph to Table and vice versa

One important design choice when storing a graph structure in a relational system is the table schema that is used. It impacts the complexity of translated queries and thus the runtime for fetching a result. Fritzsche has evaluated different approaches in [10], which are partly demonstrated in the following.

Table schema for vertices and edges The first types of entities that need to be considered are the vertices and edges themselves. In this work, we compared two commonly used schemata to represent vertex and edge datasets as relations.

The first one is the **GVE-Schema** (**G**raph **V**ertex **E**dge-Schema) or **Vertical Schema** [1,29,27]. It mainly uses two tables - one for vertices and one for edges. Each table has an identifier and a label attribute. The table used for edges has, in addition, two attributes storing the source and target vertex identifier of an edge. To represent bitemporal graphs, two additional attributes represent the lower and upper bound of the valid time period whereas two additional attributes represent the lower and upper bound of the transaction time period. The representation of properties is discussed later. To ensure the integrity of the graph, primary and foreign key constraints are used. For vertices and edges, the primary key is a combination of the identifier and the transaction period ending timestamp. The edge table has, in addition, two foreign keys pointing to the vertex table - one for the source vertex identifier and one for the target vertex identifier. One advantage of this schema is that it is flexible about new labels since labels are an attribute of the table. Another advantage is that querying for all vertices or edges can be done in one step. A disadvantage is the size of the two tables, which can be large. Also joining this kind of large tables might be an issue in terms of performance.

The second one is the **TFL-Schema** (Tables For Labels-Schema) or **Horizontal Schema** [1,29]. Here, vertices and edges are separated by their label and stored in one table per label. The specialty of the edge tables is that the node labels of the source and target nodes are also taken into account when separating. For example, there is one table of *like* edges between vertices of type *Person* and one other table of *like* edges that connect vertices of type *Person* and *Post*. This is because the foreign keys point to potentially different tables storing source and target vertices. The advantage of the TFL-Schema is the reduced table size compared to the GVE-Schema. This is particularly conceivable for answering path queries that needs joining tables. Another advantage is that queries for a specific node and edge type can simply query for the entire table,

whereas filtering is required in the GVE-Schema. A disadvantage of this schema is that all tables of nodes or edges have to be queried if no label is specified in a query. Adding nodes or edges with new labels to the graph also may require additional tables to store the corresponding element types.

To overcome the limitations of both schema models and combine their advantages, we introduce the **HyVE-Schema** (**Hy**brid Schema for **V**ertices and **E**dges). As the name suggests, this is a hybrid of the two schemes mentioned and leaves a decision open depending on the application. For each vertex and edge label, it can be decided whether the elements will be stored in a general vertex or edge table, or in a label-specific table. A metadata table stores these decisions and can be queried if the information about the location of a specific vertex and edge type is needed.

Table schema of properties Another important design decision is the modeling of properties. Again, we differentiate two ways of representing properties of vertices and edges in the relational schema.

The first one is **PAC** (**P**roperties **a**s **C**olumns), which, like the name suggests, stores properties as additional columns in the vertex and edge tables (independent of the schema for vertices and edges). The column or attribute name is defines the property key whereas the values in the entities correspond to property values. The bitemporal modeling is implicitly used from the tables containing the columns. One advantage of this approach is the usage of datatypes for property values that are supported by SQL. For example, a property `name` can be represented by an attribute with the same name and datatype `VARCHAR`. This way, a datatype may be assigned to each property. Another advantage is that no joins need to be executed when accessing the properties. If an element does not have a property, the respective column value is set to null. Disadvantages are thus many null values for elements that do not have a property that is modeled as a column. Another disadvantage is the need for change of the table schema if new property types are added, which results in the addition of a new column. The last weak point of this approach is that the whole table entry for a vertex or edge has to be copied when system-versioning is enabled. Even if only a single property value changes, a copy of the whole row is created which gets the new value and thus leads to many redundant attribute values.

The second approach is **PAT** (**P**roperties **a**s **T**able). Here, a property table per vertex or edge table stores all properties. A property table has mainly 3 attributes: the identifier of the parent element (vertex or edge), the key and the value. The bitemporal modeling is given by four additional attributes, as described in the GVE-Schema. A foreign key points from the element identifier to the identifier of the respective element table. One advantage is that all properties are optional, as the property graph model defines. It is easy to add new properties to a vertex or edge type just by inserting a new row in the property table. Another advantage is that the change of a property does not affect the vertex or edge entity itself, but just creates a copy in the property table for this property through the system-versioning. Disadvantages are the need for one join

per property access and the fixed data type for the value column, implying that all property values must have the same datatype.

Again, to combine the strengths of both strategies and to overcome some weaknesses, we introduce the **HyPe-Schema** for bitemporal modeled vertex and edge properties. For each vertex and edge type, it can be chosen whether a property is stored as a column in the vertex or edge table (see PAC), or in a separate property table for this vertex or edge type. A metadata table stores these decisions and can be queried, if the information about the location of a specific property is needed.

6.3 Graph Querying

To query a TPGM$^+$ graph that is stored in the bitemporal graph storage with T-PGQL, there are two types of queries: (1) a single query that is executed once and returns the result set back to the user (colored orange in Fig. 2) and (2) a query registration for continuous event notification (colored green in Fig. 2).

For both cases, the query string will be first parsed by a *T-PGQL Syntax Parser*, which verifies and analyses the query and creates an object representation containing lists of projections, predicates, requested vertices and edges. The parser is based on the Open-Source project *PGQL Parser and Static Query Validator* which is available on GitHub[5]. We extended this implementation by supporting all additional temporal features of the T-PGQL language. The resulting parsed query object is then used as input for the *T-PGQL-2-SQL translator*. This component creates a SQL Select query from the content of the query object in a way that the result of the query forms the T-PGQL result.

For the single query execution, the SQL Select query is executed on the relational database and the resultset is transferred back to the user that initially called the query method. For the continuous event notification, the SQL select query, which is again a representation of a transaction-world event, is given to a *Query Analyser* that extracts necessary information of the query for the registration routine, e.g., touched tables, needed attributes and predicates [34]. Depending on the type of notification (GCN or GRCN), several triggers were registered on the concerned tables that create the notifications, which are sent to a message queue. The user, who has access to this queue, now receives notifications about events that either affect the related tables (GCN) or the query result (GRCN).

7 Summary

We presented a summary of our cooperation outcomes. We introduced (1) the bitemporal property graph model *TPGM*$^+$ which supports the evolution of property values, (2) the declarative graph query language *T-PGQL* to match patterns in a TPGM$^+$ graph, (3) an event detection engine that allows the registration of

[5] `https://github.com/oracle/pgql-lang`

T-PGQL queries on a bitemporal graph storage for continuous notifications of graph changes and (4) a prototype called *BiTeGra* which stores TPGM$^+$ graphs in temporal relational tables and offers features to modify and query the graph as well as registering queries for event detection.

References

1. Adameit, T.: Evaluation des EPGM auf Basis von Apache Spark. Master thesis at Universität Leipzig (2020)
2. Angles, R.: The property graph database model. In: AMW. CEUR Workshop Proceedings, vol. 2100. CEUR-WS.org (2018)
3. Besta, M., Fischer, M., Kalavri, V., Kapralov, M., Hoefler, T.: Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. Computing Research Repository (CoRR) (2020)
4. Campos, A., Mozzino, J., Vaisman, A.: Towards temporal graph databases. arXiv preprint arXiv:1604.08568 (2016)
5. Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-varying graphs and dynamic networks. International Journal of Parallel, Emergent and Distributed Systems **27**(5), 387–408 (2012)
6. Corp., O.: Oracle's Graph Database, `https://www.oracle.com/de/database/graph/`
7. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. The VLDB Journal pp. 1–34 (2021)
8. Ediger, D., McColl, R., Riedy, J., Bader, D.A.: Stinger: High performance data structure for streaming graphs. In: 2012 IEEE Conference on High Performance Extreme Computing. pp. 1–5. IEEE (2012)
9. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: Proc. ACM SIGMOD. pp. 1433–1445 (2018)
10. Fritzsche, P.: Relationale Speicherung und Verarbeitung für temporale Graphdaten. Master thesis at Universität Leipzig (2021)
11. Holme, P., Saramäki, J.: Temporal networks. Physics Reports **519**(3), 97 – 125 (2012). https://doi.org/https://doi.org/10.1016/j.physrep.2012.03.001, `http://www.sciencedirect.com/science/article/pii/S0370157312000841`, temporal Networks
12. Honeywell: Aerospace & defense - sensors and switches product range guide, `https://sensing.honeywell.com/honeywell-sensing-aerospace-defense-rangeguide-000703-6-en.pdf`
13. Iyer, A.P., Li, L.E., Das, T., Stoica, I.: Time-evolving graph processing at scale. In: Proceedings of the fourth international workshop on graph data management experiences and systems. pp. 1–6 (2016)
14. Jensen, C.S., Snodgrass, R.T.: Temporal data management. IEEE Trans. Knowl. Data Eng. **11**(1), 36–44 (1999). https://doi.org/10.1109/69.755613, `https://doi.org/10.1109/69.755613`
15. Johnston, T.: Bitemporal data: theory and practice. Newnes (2014)
16. Kankanamge, C., Sahu, S., Mhedbhi, A., Chen, J., Salihoglu, S.: Graphflow: An active graph database. In: Proceedings of the 2017 ACM International Conference on Management of Data. p. 1695–1698. SIGMOD '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3035918.3056445, `https://doi.org/10.1145/3035918.3056445`

17. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). pp. 997–1008. IEEE (2013)
18. Kostakos, V.: Temporal graphs. Physica A: Statistical Mechanics and its Applications **388**(6), 1007–1023 (2009)
19. Kulkarni, K., Michels, J.E.: Temporal features in SQL:2011. ACM Sigmod Record **41**(3), 34–43 (2012)
20. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: Llama: Efficient graph analytics using large multiversioned arrays. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 363–374. IEEE (2015)
21. Neo4j Inc.: Cypher Query Language (July 2020), `https://neo4j.com/developer/cypher/`
22. Oracle: Continuous Query Notification (CQN), `https://cx-oracle.readthedocs.io/en/latest/user_guide/cqn.html`
23. Oracle: PGQL - Property Graph Query Language, `https://pgql-lang.org/`
24. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems. pp. 1–6 (2016)
25. Rodriguez, M.A.: The gremlin graph traversal machine and language (invited talk). In: Proceedings of the 15th Symposium on Database Programming Languages. pp. 1–10 (2015)
26. Rodriguez, M.A., Neubauer, P.: Constructions from Dots and Lines. arXiv:1006.2361v1 (2010)
27. Rost, C., Gomez, K., Täschner, M., Fritzsche, P., Schons, L., Christ, L., Adameit, T., Junghanns, M., Rahm, E.: Distributed temporal graph analytics with GRADOOP. The VLDB Journal pp. 1–27 (2021)
28. Rost, C., Thor, A., Rahm, E.: Analyzing temporal graphs with GRADOOP. Datenbank-Spektrum **19**(3), 199–208 (2019)
29. Saalmann, E.: Relationale Abstraktion des EPGM unter Verwendung der Apache Flink Table-API. Master thesis at Universität Leipzig (2019)
30. Sengupta, D., Sundaram, N., Zhu, X., Willke, T.L., Young, J., Wolf, M., Schwan, K.: Graphin: An online high performance incremental graph processing framework. In: European Conference on Parallel Processing. pp. 319–333. Springer (2016)
31. Sevenich, M., Hong, S., van Rest, O., Wu, Z., Banerjee, J., Chafi, H.: Using domain-specific languages for analytic graph databases. Proceedings of the VLDB Endowment **9**(13), 1257–1268 (2016)
32. Steer, B., Cuadrado, F., Clegg, R.: Raphtory: Streaming analysis of distributed temporal graphs. Future Generation Computer Systems **102**, 453–464 (2020)
33. Wu, H., Cheng, J., Huang, S., Ke, Y., Lu, Y., Xu, Y.: Path problems in temporal graphs. Proceedings of the VLDB Endowment **7**(9), 721–732 (2014)
34. Zimmer, M.: Kontinuierliche Graph-Query Notifikationen auf Basis eines RDBMS. Bachelor thesis at Universität Leipzig (2021)