Journal of Intelligent Information Systems (2023) 61:569–609
https://doi.org/10.1007/s10844-023-00780-6

Modeling and querying temporal RDF knowledge graphs with relational databases

Ruizhe Ma[1] · Xiao Han[2] · Li Yan[2] · Nasrullah Khan[2] · Zongmin Ma[2,3]

Received: 4 November 2022 / Revised: 29 January 2023 / Accepted: 30 January 2023 /
Published online: 30 March 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract

RDF (Resource Description Framework), a standard resource description model, is popularized and applied in many application scenarios for its explicit representation of semantics. To represent and process time-aware semantics with RDF, the temporal RDF model is proposed and applied in temporal knowledge graphs. The requirement for efficiently handling diverse temporal RDF data has become increasingly important with the rapid development and popularity of RDF. In this paper, we propose a novel temporal RDF model and effectively tackle the management of temporal RDF data in relational databases. In particular, we propose a temporal RDF model called tRDF to represent both temporal entities and relationships and further propose a temporal query language for the tRDF model. To manage temporal RDF data in an effective manner, we propose to store temporal RDF data with relational databases that follow the SQL:2011 standard and support temporal data manipulation. To query the tRDF data stored in relational databases with the tRDF query language, we implement the transformation from this query language to SQL. The experimental results show the feasibility and effectiveness of the proposed tRDF model as well as its storage and query methods.

Keywords Temporal RDF model · Relational databases · Data persistence · Querying

1 Introduction

RDF[1] is a metadata model recommended by the W3C (World Wide Web Consortium), which can explicitly describe resources on the Web and the relationships between these resources. RDF has good machine readability, and its syntactic form is very similar to the composition of

[1] http://www.w3.org/RDF/

Xiao Han and Ruizhe Ma are co-first authors and contributed equally to this study.

✉ Zongmin Ma
zongminma@nuaa.edu.cn

1 Richard A. Miner School of Computer & Information Sciences, University of Massachusetts Lowell, Lowell, MA 01854, USA

2 College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China

3 Collaborative Innovation Centre of Novel Software Technology and Industrialization, Nanjing 210023, China

knowledge. Thus, the RDF model has been extensively accepted as a representation model of knowledge graphs, which was formally introduced by Google in 2012 with the aim of improving the performance of search engines. Nowadays, knowledge graphs have been widely applied in diverse domains, and many knowledge graphs have become available (e.g., DBpedia[2] and Wikidata[3]). With the increasing scale of knowledge graphs, efficient storage and query of the huge amount of RDF data are of crucial importance. Traditionally, there are three main categories of RDF storage methods, which are memory-based method (Atre & Hendler, 2009), disk-based (Wu et al., 2009), and database-based [3, 9, 11, 26, 35, 40- 42, 45, 51, 52], respectively. Among them, the database storage method has become the primary means to manage large-scale RDF data because of the mature techniques and numerous products of database systems (Ma et al., 2016). It is especially true for relational databases.

The real world is dynamic, and any individual may change from time to time. Data with temporal features are known as temporal data. One can find temporal data available in many fields (e.g., geographic information systems, weather forecasts, dynamic social networks, the Internet of Things, etc.). The issue of representing and managing temporal data has been investigated in the context of relational databases for a longtime [10, 43, 44]. After realizing the importance and urgency of explicitly manipulating temporal data in relational databases in a normal way, the ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) jointly published SQL:2011, which is the most recent revision of the SQL (Structured Query Language) standard, replacing SQL:2008. One of the new features in SQL:2011, and also the most important new feature in SQL:2011, is the ability to create and manipulate temporal tables, in which rows are associated with one or more time periods (Kulkarni & Michels, 2012). In addition to the temporal relational database model, several emerging temporal data models have been proposed for dealing with temporal data in recent years. To represent and share temporal data on the Web, for example, the temporal XML model is proposed and applied (Faisal & Sarwar, 2014); also, to manage and process big temporal data, temporal NoSQL databases (e.g., column-oriented NoSQL databases) are proposed (Chen, et al., 2022).

In the context of RDF, the classical RDF model can only represent static semantics, i.e., the current state of resources and their relationships. To capture the dynamic state of resources and their relationships, several temporal RDF models have been proposed by extending the static RDF model. Basically, we can identify three major types of temporal RDF models, which are the temporal RDF model for version control (Gutierrez et al., 2007), the temporal RDF model with time label (Pugiles et al., 2008), and the temporal RDF model with the triple extension (Koubarakis & Kyzirakos, 2010). Among these temporal RDF models, the temporal RDF model with a time label is widely accepted and used as it does not change the structure and extensibility of current RDF triples. In the current temporal RDF models, a temporal RDF triple contains a timestamp that is attached to either the predicate of the RDF triple or the whole RDF triple. The RDF triple with a temporal granularity of the whole triple declares a temporal statement (i.e., a temporal fact), but it is unclear which one of the subject, predicate, and object in this triple is actually time-aware. Also, the RDF triple with a temporal granularity in the predicate of triple clearly indicates a time-aware predicate, but it fails to represent temporal information in the object of the triple. It is possible and common for the same subject and predicate to have several different objects over time. At this point, it is essential to explicitly represent time-aware objects in temporal RDF triples.

2 https://www.dbpedia.org/

3 https://www.wikidata.org

For the current temporal RDF models, few efforts are devoted to temporal RDF query (e.g., (Tappolet & Bernstein, 2009; Zaniolo et al., 2018)) and temporal RDF index (e.g., (Pugiles et al., 2008; Yan et al., 2019)). Although storing the classical RDF in databases has been widely investigated and applied, to the best of our knowledge, there is no work investigating temporal RDF storage. The temporal RDF has been used in knowledge graphs to represent temporal knowledge [50, 21]. Recently, temporal knowledge graphs are receiving increasing attention for their representation learning (e.g., (Chen, et al., 2022)), but they are mainly based on the temporal triples with a temporal granularity of the whole triple. With the widespread application of knowledge graphs in diverse time-sensitive domains (e.g., the Internet of Things), a huge amount of temporal RDF data is being proliferated and becoming available. Therefore, it is increasingly important to propose a more semantically expressive temporal RDF model and then efficiently manage large-scale temporal RDF data.

In this paper, we propose a new temporal RDF model named tRDF by exploiting the time label, which can be applied to the predicate and/or object of a triple as a timestamp. Clearly, the tRDF model is different from the existing temporal RDF models whose time labels are attached to either the predicates of the RDF triples or the whole RDF triples. We present the syntax and semantics of the tRDF model. Based on the tRDF model, we particularly advocate storing temporal RDF data in the temporal relational databases and, with this, propose mapping rules to map the tRDF model to the temporal relational tables. To query the temporal RDF data that is actually stored in the temporal relational databases, we formalize a temporal SPARQL (Simple Protocol and RDF Query Language) for the tRDF model, termed as tSPARQLt, and then provide the transformation of partial queries from tSPARQLt to SQL, a standard query language for the relational databases. We validate our proposed model and approach through comparative experiments. Although there are some proposals for temporal metrics, to the best of our knowledge, this paper is the first effort to model and query temporal RDF data with temporal relational databases.

The rest of the paper is organized as follows. Section 2 provides a brief overview of related work in RDF storage, temporal databases, and temporal RDF. Section 3 presents some preliminaries. Section 4 proposes a new temporal RDF model named tRDF and provides its syntax and semantics. Section 5 presents mapping rules and algorithms for tRDF data storage in relational databases. Section 6 formalizes a query language, called tSPARQLt, for the temporal tRDF model and defines the rules for transforming partial queries of tSPARQLt to SQL queries. Section 7 presents the experimental evaluations for our proposed storage and query method. Section 8 concludes this paper.

2 Related Work

In this section, we present the related work in three categories: RDF storage in relational databases, temporal databases, and temporal RDF.

2.1 RDF storage

Data storage is the foundation of implementing data management. To manage large-scale RDF data, many proposals have been developed to store RDF data, which are

roughly categorized as *memory-based storage* (Atre & Hendler, 2009), *disk-based storage* (Wu et al., 2009), and *database-based storage* [3, 9, 11, 26, 35, 40- 42, 45, 51, 52]. The memory-based and disk-based storages load RDF data as triples directly into memory and store RDF triples on the local hard disk, respectively, which are collectively referred to as a local storage approach. The local storage approach preserves the triadic structure and semantics of RDF triples well, but it also suffers from several drawbacks. The memory-based storage is clearly limited by the size of memory and is only applicable to storing a small amount of RDF data. The disk-based storage shifts data storage place from memory to disk and thereby satisfies the requirement of storing larger-scale RDF data. The disk-based RDF storage is a category of *native stores* (e.g., RDF-3X (Neumann & Weikum, 2008)) that use customized binary RDF data representation and are built directly on the file system (Bornea, et al., 2013). Note that the native RDF stores fail to provide full and strong support for data access and control management. Database management systems (DBMSs) are designed especially for efficient data storage and management. At this point, non-native RDF stores built on top of existing database management systems (DBMSs) become the mainstream method of RDF data storage (Ma et al., 2016).

Relational databases (RDBs) have been widely used for their solid theoretical foundation and strong technical support in products as well as development tools. Typically, there are three common ways to store RDF data with relational databases (Ma et al., 2016): *vertical stores*, *horizontal stores,* and *type stores*. The vertical stores (e.g., Triplet (Wolff et al., 2015)), also known as *triple stores*, create a single relational table with three columns, and each RDF triple is directly mapped into a tuple of the relational table. Here the subject, predicate, and object of an RDF triple become three attribute values of the corresponding tuple. The horizontal stores (e.g., C-Store (Stonebraker, et al., 2005), Virtuoso (Erling & Mikhailov, 2009), and SW-store (Abadi et al., 2009)) create either a single relational table that contains all predicates of RDF triples as the table's column names or a set of relational tables and a relational table only contains one predicate as the table's column name. Note that, in the horizontal stores, the created relational table also contains a column name representing the subject of RDF triples in addition to the column name(s) from the predicate(s). In the horizontal stores with a single relational table, the RDF triples with the same subject and becomes a tuple of the relational table. In the horizontal stores with a set of relational tables, the RDF triples with the same predicate appear in the same relational table, where each RDF triple corresponds to a tuple of the relational table. The type stores (e.g., RDFBroker (Sintek & Kiesel, 2006), RDB2RDF(Salas, et al., 2011), and Jena (McBride, 2002)) may create multiple relational tables, and each one is for a type of subject, in which a relational table contains the properties as n-ary table columns for the same subject. In addition to the three basic relational stores of RDF above, there are also efforts in RDF data storage that use two or more of the three basic stores concurrently or revise the three basic stores (e.g., (Bornea, et al., 2013)).

For large-scale RDF data management, it is essential to ensure the scalability of RDF stores by using optimization structures such as indexes and data partitioning. In (Weiss et al., 2008), an RDF storage scheme called Hexastore was proposed, which enhances the vertical partitioning idea and takes it to its logical conclusion. As a result, a sextuple indexing scheme was applied in Hexastore. Unlike Hexastore, which builds exhaustive indexing of pairs of positions in triples, RDF-3X (Neumann & Weikum, 2008) builds exhaustive indexing of all permutations of triple positions, and TripleT (Wolff et al., 2015) builds exhaustive indexing of all single positions. In addition, to significantly improve the

scalability of massive RDF stores, distributed/parallel RDF stores have been developed (Papailiou, et al., 2013). In contrast to centralized RDF stores that are single-machine solutions, distributed RDF stores (e.g., 4store[4]) partition triples across multiple machines and parallelizes query processing (Ma et al., 2016). In distributed RDF stores, RDF data partitioning is a crucial issue. Distributed RDF stores adopt two categories of partitioning: *horizontal partitioning* and *hash partitioning,* according to how the RDF data are partitioned and how partitions are stored for access (Lee & Liu, 2013). Horizontal partitioning generally partitions an RDF dataset across multiple servers by using horizontal (random) partitioning, where the partitions are stored by using distributed file systems (e.g., HDFS (Hadoop Distributed File System)[5]), and then queries are processed by parallel access to the clustered servers by using distributed programming models (e.g., Hadoop MapReduce). Hash partitioning partitions an RDF dataset across multiple nodes by using hash partitioning on three components of RDF triples (i.e., *subject*, object, and *predicate*) or any combination of them, where the partitions are locally stored in a database like HBase or an RDF store like RDF-3X and then accessed through a local query interface.

To deal with big data, NoSQL databases have emerged as a new infrastructure for massive data storage and management. As a result, NoSQL databases are applied to handle massive RDF data (Cudre-Mauroux, et al., 2013). Moreover, several NoSQL stores for RDF management (e.g., RDFChain (Choi et al., 2013), the Jena-HBase (Khadilkar, et al., 2012), and Trinity.RDF (Shao et al., 2013)) have been proposed. RDF data management merits the use of NoSQL databases because of their scalability and high performance. Viewed from the theoretical foundation and technical support in products as well as development tools, however, relational databases are still in a dominant position for a relatively long period of time. Concerning massive RDF data stored in relational and NoSQL databases, one can refer to a comprehensive review in Ma et al. (2016). Note that the existing approaches for RDF data stores, both in relational and NoSQL databases, cannot explicitly deal with temporal RDF data.

2.2 Temporal databases

Temporal data representation and management have been widely investigated in the context of relational databases. As early as the 1980s, the temporal relational model was proposed by including temporal columns in the relational model. In (Clifford & Croker, 1987), a historical relational model was proposed, in which several issues like relation, tuple, and field value with temporal information were discussed. Time in temporal relational databases can be classified into three types (Mckenzie & Snodgrass, 1991; O'Connor & Das, 2010): *valid time*, *transaction time,* and *user-defined time*. The temporal relational databases containing only valid time are called the historical relational databases, the temporal relational databases containing only transaction time are called the rollback relational databases, and the temporal relational databases containing both valid time and transaction time are called the bi-temporal relational databases. With the development of temporal relational models, several query languages have been proposed. TQuel, proposed by Snodgrass in (Snodgrass, 1987), a well-known temporal query language, is an extension of Quel, which is upward compatible with Quel, which is very helpful in promoting the temporal data model and the temporal query language. Snodgrass further proposed a temporal

[4] http://4store.org/

[5] http://hadoop.apache.org/hdfs

query language, TSQL2, in (Snodgrass, 1994). TempSQL in Gadia (1988) is a temporal relational model which provides a complete temporal query language. These proposed temporal query languages support both valid time and transaction time.

SQL:2011 should be a milestone in the research and development of temporal relational databases. As the latest edition of the SQL standard published by the ISO/IEC in 2011, SQL:2011 explicitly provides support for creating and manipulating temporal data with temporal tables (Kulkarni & Michels, 2012). In SQL:2011, a common column may be related to an application-time period, a system-time period, or both. Furthermore, a time (application- or system-) period contains the period start time and the period end time, which are declared as two special columns named by the user. After SQL:2011 was published, many efforts have been made to extend traditional database management systems. Gao et al. in (Gao, et al., 2018), for example, proposed a new framework for the design of temporal relational databases, which supports effective access to current and historical information; Lu et al. in Lu et al. (2019) contributed to temporal extensions in distributed database management systems so that the efficiency of managing temporal data can be improved.

With the increased use of NoSQL databases for big data management, few efforts are devoted to dealing with temporal big data. Hu and Dessloch (Hu & Dessloch, 2015) proposed to use column-oriented NoSQL databases (CoNoSQLDBs) for temporal data management and processing. In the context of spatio-temporal data, Zhong et al. (Zhong et al., 2013) combined NoSQL databases and Hadoop to achieve the storage of temporal data based on distributed techniques, and Fox et al. in (Fox, et al., 2013) used the high scalability of NoSQL databases to achieve high performance queries on temporal data. Unlike SQL:2011, the current NoSQL databases do not support temporal data management. Few temporal extensions to NoSQL databases are designed for temporal RDF data store. At this point, it is a good choice to apply SQL:2011 for modeling and processing temporal RDF data, just like the relational databases for the common RDF data.

2.3 Temporal RDF models and query languages

It is recognized that in many practical applications of RDF, it is necessary to attach triples metadata into RDF (Hogan, et al., 2010). In (Udrea et al., 2010), annotated RDF model was formally proposed, where RDF triples are annotated by members of a partially ordered set. For the annotated RDF in Udrea et al. (2010), a general extension to RDF Schema (RDFS) was proposed in Straccia, et al. (2010), and a query language AnQL was then developed for the annotated RDFS in Lopes, et al. (2010). Annotations in RDF can support several specific domains to represent the temporal aspects, uncertainty, trust, and provenance of the RDF triples. Temporal RDF models are explicitly proposed to handle semantically metadata with temporal information. Gutierrez et al. in (Gutierrez et al., 2007) proposed a temporal RDF model by using a version control approach. They presented the syntax and semantics of the proposed temporal RDF model. In (Pugiles et al., 2008), a temporal RDF model was proposed by adding timestamps to RDF predicates, and the concept of indeterminate temporal triples was introduced. Koubarakis et al. (Koubarakis & Kyzirakos, 2010) proposed the quadtuple structure with temporal information using the triple extension method. Among the above temporal RDF models, the time label method is widely used because it preserves the original triple structure RDF. In (Grandi, 2009), a temporal RDF model was proposed, which uses triple timestamping with temporal elements. The data model is equipped with manipulation operations to manage temporal versions of an ontology. A

survey of temporal extensions to RDF is provided in Wang and Tansel (2019), in which the proposals for extending RDF for modeling temporal data are classified into explicit reification or implicit reification according to the used reification. The time investigated in the existing temporal RDF models mainly focuses on valid time.

Ontologies can be seen as a formal representation of knowledge over RDF data. In addition to the RDF with time, several studies have been proposed to manage domain knowledge evolution in the context of ontology versioning. In (Brahmia, et al., 2022), temporal versioning of both ontology instances and ontology schemas was considered, where ontology schema changes are triggered by non-conservative updates to ontology instances. That is, an ontology schema versioning is driven by instance updates. To address the problem of asynchronous versioning in the context of a materialized integration system, the principle of ontological continuity was proposed in Xuan et al. (2006) to support ontology changes. With the proposed principle, each old instance can be managed by using the new version of the ontology. Focusing on time-sensitive application domains, Canito, Corchado, and Marreiros (Canito et al., 2022) systematically reviewed the state of the art of representation of time and ontology evolution in the predictive maintenance field. It was identified that, although ontologies have many applications in predictive maintenance, there have been few studies on ontology evolution, and applications of time to the problem of ontology evolution are still in the open.

To query temporal RDF data, a temporal SPARQL language called τ-SPARQL was proposed in (Tappolet & Bernstein, 2009) for temporal RDF graph, where τ-SPARQL queries can be translated to standard SPARQL queries. A temporal extension of SPARQL was presented in Grandi (2010), which was aimed at embedding several features of the TSQL2 consensual language. In (Zaniolo et al., 2018), a point-based temporal extension of SPARQL, called $SPARQL^T$, was proposed for the main-memory RDF-TX system, which supports user-friendly by-example temporal queries on historical knowledge bases derived from Wikipedia. Based on classical OBDA (ontology-based data access) systems, Brandt et al. in Brandt, et al. (2017) proposed a framework of temporal OBDA, which can extract information about temporal events in RDF format and provides a SPARQL-based query language for retrieving temporal information. Concentrating on the OBDA system for query answering with temporal data and ontologies, Elem et al. in (Kalayci, et al., 2018) developed a tool called Ontop-temporal, which can access timestamped log data. To further improve the efficiency of querying large-scale temporal RDF data, few efforts have worked on indexing temporal RDF. In (Pugiles et al., 2008), an index structure named tGRIN was proposed, which builds a specialized index for temporal RDF triples stored in Jena2, Sesame, and 3store. An index structure was proposed in (Zaniolo et al., 2018) for the original temporal RDF graphs, where the prefix path index for querying subjects of temporal RDF triples and the suffix index for querying objects of temporal RDF triples were built, respectively. In addition to several temporal SPARQL languages, few proposals for spatiotemporal SPARQL languages have been developed. In (Perry et al., 2011), for example, SPARQL was extended to SPARQL-ST so that spatiotemporal queries can be supported, and in Koubarakis and Kyzirakos (2010), the query language stSPARQ was developed for spatiotemporal RDF model in the context of the Semantic sensor Web.

The RDF model has been applied to the infrastructure of knowledge graphs (Hogan et al., 2022). With temporal RDF triples, some recent efforts have been made to investigate temporal knowledge graphs (Huang et al., 2020; Lu et al., 2019). There has been an increasing interest in learning representations of temporal knowledge graphs (e.g., (Chen, et al., 2022; Zhu, et al., 2021)). As of yet, no proposals exist for

storing and querying multigranularity temporal RDF data using temporal relational databases.

3 Preliminaries

In this section, we introduce some preliminaries about the RDF model, SPARQL (Simple Protocol and RDF Query Language), and SQL:2011.

3.1 RDF Model

RDF is a metadata model proposed by W3C to describe Web resources and their mutual relationships. It provides a general framework for the description and interaction of information. An RDF model is described with a set of RDF triples. An RDF triple in the form of (*subject*, *predicate*, *object*) (abbreviated to (S, P, O)) is a statement in which the *subject* is the resource being described, the *predicate* is the property being described with respect to the resource, and the *object* is the value for the property. Here, a resource is anything with a *URI* (Universal Resource Identifier), and an object is a literal (if the corresponding *predicate* is an attribute of the resource) or another resource (if the corresponding *predicate* is a relationship of resources).

Definition 1 (RDF model) An RDF model is a set of triples, and an RDF triple is formally defined as $(S, P, O) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I, B, and L are infinite sets of *IRI*s (Internationalized Resource Identifiers), blank nodes and RDF literals, respectively.

RDF model can be described with several formats such as RDF/XML, N-Triples, Turtle, RDFa (Resource Description Framework in Attributes), and JSON-LD (JSON for Linking Data). RDF/XML represents RDF data by using the syntax of XML. Since the diffused syntax format of RDF/XML is too complex to understand, N-Triples (NT) are applied to represent RDF data, which is most similar to the syntax of the RDF model and easy to read and parse. Nowadays, many public RDF datasets (e.g., Wikidata and DBpedia) are published in the format of N-Triples. In addition, Turtle is an optimization of RDF/XML, which makes the representation more compact by indicating the prefix; RDFa uses HTML5 to represent RDF data; JSON-LD uses key-value pairs to describe RDF data. Also, the RDF model can be represented as a directed and labeled graph, where subjects and objects of triples are the vertices and predicates of triples are the edges from subject vertices to object vertices.

In addition to its syntax, an RDF model has its semantic interpretation.

Definition 2 (RDF model semantics) For an RDF model, its semantic interpretation I consists of the following elements:

(1) A non-empty set of resources (i.e., IR) is called the domain of I.
(2) A set IP is called the set of properties of I.
(3) A set IL is called the set of literals that contains all the objects of the literal type.
(4) A mapping $IEXT$ from IP into the power-set $IR \times IR$ (i.e., the set of sets of $<x, y>$ pairs with x and y in IR).
(5) A mapping IS from $IRIs$ into $(IR \cup IP)$.

(6) A partial mapping *ILR* from literals into *IR*.

3.2 SPARQL

SPARQL[6] recommended by W3C, is a query language for RDF data. With a simple query statement structure, SPARQL is easy to understand and read, and has a reasoning ability to optimize the query efficiency of RDF. Considering the essential graphic structure of the RDF model, SPARQL queries evaluate the user's requirements against RDF datasets in a graph matching way. A statement of SPARQL query generally consists of four components, which are *query form*, *dataset*, *graph pattern with constraints*, and *solution modifier*, respectively.

Four query forms in SPARQL can be identified as follows.

- SELECT: identifies and returns the matched datasets or graphs.
- CONSTRUCT: creates a new RDF graph.
- ASK: judges whether the RDF graph has the result of a given query.
- DESCRIBE: returns information of all graph nodes matched by the query.

Among these four query forms, SELECT is widely used for searching RDF. A SELECT query has the basic structure of *SELECT-FROM-WHERE*. The *SELECT* clause indicates the set of variables being shown in query answers, and a dataset in the *FROM* clause is used to specify the RDF data to be queried. A graph pattern in the *WHERE* clause describes the user's query requirement as a filter condition. We can identify three major kinds of graph patterns: the *basic graph pattern*, the *group graph pattern,* and the *optional graph pattern*. The basic graph pattern (BGP) consists of a number of triple patterns separated by ".". A triple pattern is a special kind of triple, where at least its subject, predicate, or object is represented by a variable. In SPARQL, a variable is introduced using "?" or "$" as a prefix. We can identify triple patterns like (S, P, ?O), (S, ?P, ?O), (S, ?P, O), (?S, ?P, O), (?S, P, O), (?S, P, ?O) and (?S, ?P, ? O). A basic graph pattern is a set of triple patterns surrounded by "{}," and all of them have to be matched in query evaluation. A group graph pattern (GGP) consists of a set of BGPs, and all of these BGPs need to be matched in query evaluation. An optional graph pattern (OGP), starting with the keyword *OPTIONAL,* is followed by one or more BGPs, which should be optional and not be requested for a mandatory match on query evaluation. In the graph patterns, the keyword FILTER can be used to explicitly filter out the set of eligible results. Also, SPARQL provides several modifiers (such as LIMIT, OFFSET, ORDER BY, etc.) to arrange the query results so that users can better view the result set.

3.3 SQL:2011

SQL is a standard query language for relational databases. SQL:2011 published by the ISO/IEC in 2011, is the latest edition of the SQL standard. SQL:2011 replaces the previous edition SQL:2008 and contains many new features, in which the most important new feature is that it can explicitly represent and deal with temporal data with temporal tables.

In SQL:2011, time periods are explicitly defined and associated with the rows of a table. A time period is demarcated by *a start time* and *an end time*. Here a period definition is a named table component, which actually identifies a pair of columns to

[6] https://www.w3.org/TR/sparql11-query/

capture the start time and the end time of the period. Note that the start column and the end column of the period are special columns in the table. Note that SQL:2011 adopts a left-closed-right-open period model to define a time period like [*a start time*, *an end time*), which includes the start time, but excludes the end time.

SQL:2011 distinguishes two types of time periods: the *system-time period* for transaction time support; an *application-time period* for valid time support. In an application-time period table, SQL:2011 applies the keywords *PERIOD FOR* to define an application-time period with a user-defined name, which contains two user-named columns to respectively represent the start time and end time of the period. Note that the period start and end columns must have the same data types, either DATE or a timestamp type. Assume that the user would create an application-time period table, *atTable*, which contains a period definition with a user-defined name, *atPeriod*. This period contains two columns with user-defined names, *atStart* and *atEnd*, which act as the start and end columns of the period. Then this temporal table is formally defined as follows.

```
CREATE TABLE atTable (
...
atStart DATE,
atEnd DATE,
...
PERIOD FOR atPeriod (atStart, atEnd)
)
```

SQL:2011 also uses the regular query syntax SELECT-FROM-WHERE to query application-time period tables. Here SQL:2011 provides several period predicates to express conditions that involve periods, including CONTAINS, OVERLAPS, EQUALS, PRECEDES, SUCCEEDS, IMMEDIATELY PRECEDES, and IMMEDIATELY SUCCEEDS.

In the system-versioned tables, SQL:2011 uses the keywords *PERIOD FOR SYSTEM_TIME* to define a system-time period with the standard-specified name: SYSTEM_TIME. The declared system-time period contains two user-named columns to respectively represent the start and end columns of the SYSTEM_TIME period. Also, the period start and end columns must have the same data types, either *DATE* or a timestamp type. However, in practice, the *TIMESTAMP* type with the highest fractional seconds precision is applied as the data type for the system-time period start and end columns. Note that the system-versioned table includes the keywords WITH SYSTEM VERSIONING in its definition. Assume that the user would create a system-versioned table, *svTable*, which contains a period definition with the standard-specified name SYSTEM_TIME, and the keywords WITH SYSTEM VERSIONING. This period contains two columns with user-defined names, *svStart* and *svEnd*, which act as the start and end columns of the period. Then this temporal table is formally described as follows.

```
CREATE TABLE svTable (
...
svStart TIMESTAMP(12) GENERATED ALWAYS AS ROW START,
svEnd TIMESTAMP(12) GENERATED ALWAYS AS ROW END,
...
PERIOD FOR SYSTEM_TIME (svStart, svEnd)
) WITH SYSTEM VERSIONING
```

SQL:2011 first provides three syntactic extensions for retrieving the table content as of a given time point or between any two given time points from system-versioned tables. The first extension is the FOR SYSTEM_TIME AS OF syntax for querying the table content as of a specified time point; The second and third extensions allow for retrieving the content of a system-versioned table between any two time points. If a query on system-versioned tables is not one of the above three syntactic options, this query specifies FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP by default, which returns the current system rows as the result only.

A temporal relational schema can be formally defined as $R=(A_1, tsA_1, teA_1, A_2, tsA_2, teA_2, \ldots, A_n, tsA_n, teA_n)$, where $A_1, A_2, \ldots, A_n$ are common attributes, and each one of them (say A_i $(1 \leq i \leq n)$) may have two associated attributes (tsA_i and teA_i), representing that A_i contains a period (application-time or system-time) with the start and end columns tsA_i and teA_i. Relational instance of R written by $r(R)$ is a set of tuples, and we have $r(R)=\{t_1, t_2, \ldots, t_m\}$. A tuple of $r(R)$, say t_j $(1 \leq j \leq m)$, is formally represented as $t_j = < a_{j1}, {}_{ts}a_{j1}, {}_{te}a_{j1}, a_{j2}, {}_{ts}a_{j2}, {}_{te}a_{j2}, \ldots, a_{jn}, {}_{ts}a_{jn}, {}_{te}a_{jn} >$, where $t_j[A_i]=a_{ji}$, $t_j[tsA_i]={}_{ts}a_{ji}$ and $t_j[teA_i]={}_{te}a_{ji}$. Here $t_j[X]$ means the value of tuple t_j on column X.

In SQL:2011, a table may be both a system-versioned one and an application-time period one, forming a so-called *bitemporal table*. Rows in bitemporal tables are associated with both the system-time period and the application-time period. Concerning temporal information in the RDF model, in this paper, we pay attention only to the application-time period in RDF and do not consider the system-time period and bi-temporal periods.

4 Temporal RDF Model

Most of the temporal RDF models proposed only attach timestamps directly to the predicates of RDF triples or the whole RDF triples. Such temporal RDF models fail to represent the temporal objects of RDF triples. In this section, we propose a novel temporal RDF model termed tRDF.

4.1 Overview of the tRDF model

First, we adopt the left-closed-right-open time model [*Ts,Te*) proposed in SQL:2011, where *Ts* and *Te* are the start time and end time of the time period, respectively. As a

special case, a time interval can be a time point, where the end time of the time period is set to be the highest value of the data type. For example, [1885–05-18, 9999–12-31) means a time point 1885–05-18. For an RDF triple, its predicate or object may be added with a time period. In the paper, we identify two types of temporal RDF triples: the temporal period is attached to the predicate to indicate a time-aware relationship between two resources when the object is a resource; the temporal period is attached to the object to indicate the time-aware value of resource on the property when the object is a literal. The RDF model with the above two types of temporal RDF triples is referred to as tRDF in this paper. We illustrate our tRDF model with examples.

Table 1 presents a classical RDF model containing 6 triples about personal information. Moreover, the graph representation of this RDF model is presented in Fig. 1, where prefixes are not shown in the figure.

As a temporal extension to the traditional RDF model given in Table 1, the tRDF model is shown in Table 2. Its graph representation is presented in Fig. 2, where prefixes are not shown in the figure.

It can be seen that the tRDF model is based on a time label, so the tRDF model only needs to modify the timestamps of some temporal triples when temporal information changes. In Table 2, for example, it is assumed that the name of Márton Garas was changed to *NameB* on January 1, 1900. Then the original temporal triple (Márton_Garas, name, Márton Garas [1885–05-18, 1930–06-26)) should be modified to (Márton_Garas, name, Márton Garas [1885–05-18,1889–12-31)) and meanwhile, a new triple (Márton_Garas, name, *NameB* [1900–01-01,1930–06-26)) should be added. Of course, it is possible that the name of Márton Garas was changed back to the original later on.

Now let us look at how to represent temporal information with two existing temporal RDF models. With the temporal RDF model whose time labels are attached to the whole RDF triples, we have temporal triples (Márton_Garas, name, Márton Garas) [1885–05-18, 1930–06-26), (Márton_Garas, gender, Male)[1885–05-18, 1930–06-26), (Márton_Garas, birthPlace, Novi_Sad)[1885–05-18, 9999–12-31) and (Márton_Garas, deathPlace, Budapest)[1930–06-26, 9999–12-31). With the temporal RDF model whose time labels are attached only to the predicates of triples, we have temporal triples (Márton_Garas, name[1885–05-18, 1930–06-26), Márton Garas), (Márton_Garas, gender[1885–05-18, 1930–06-26), Male), (Márton_Garas, birthPlace[1885–05-18, 9999–12-31), Novi_Sad) and (Márton_Garas, deathPlace[1930–06-26, 9999–12-31), Budapest). Although these two models can model temporal information in triples, their semantics are ambiguous. With our tRDF model, we have temporal triples (Márton_Garas, name, Márton Garas [1885–05-18, 1930–06-26)), (Márton_Garas, gender, Male [1885–05-18, 1930–06-26)), (Márton_Garas, birthPlace[1885–05-18, 9999–12-31), Novi_Sad) and (Márton_Garas, deathPlace[1930–06-26, 9999–12-31), Budapest). Clearly, they can more exactly describe the temporal semantics in real-world scenarios.

4.2 tRDF syntax

4.2.1 tRDF triple

In the tRDF model, time labels are applied as timestamps, which are added to the predicates or the objects of the common RDF triples, depending on the type of objects. The triples with timestamps in their predicates or the objects are referred to as temporal triples in this paper. The syntax of the tRDF model is declared as a set of temporal triples.

Table 1 An example of a traditional RDF model

Subject	Predicate	Object
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/birthDate	"1885–5-18"^^<http://www.w3.org/2001/XMLSchema#date>
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/birthPlace	http://dbpedia.org/resource/Novi_Sad
http://dbpedia.org/resource/Márton_Garas	http://xmlns.com/foaf/0.1/name	"Márton Garas"@en
http://dbpedia.org/resource/Márton_Garas	http://xmlns.com/foaf/0.1/gender	"male"@en
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/deathDate	"1930–6-26"^^<http://www.w3.org/2001/XMLSchema#date
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/deathPlace	http://dbpedia.org/resource/Budapest

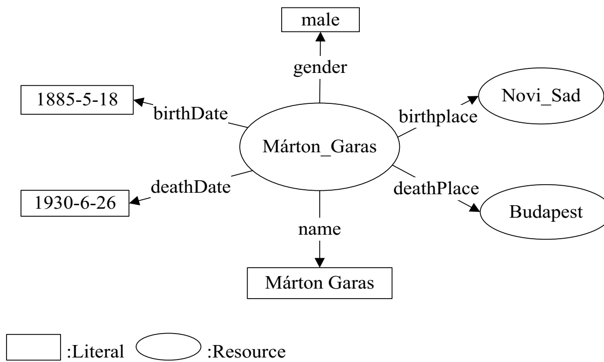

Fig. 1 An example of an RDF graph

Following the step of SQL:2011, a time period for a timestamp is uniformly expressed as [*Ts, Te*), where *Ts* and *Te* are the start time and end time of the time period, respectively. Here two cases are considered: $Ts = Te$ (the time interval is a time point) and $Ts < Te$ (the time interval is truly a period of time).

Definition 1 (tRDF triple) Temporal triples in the tRDF model have the form of (*S,P*[*Ts,Te*)*,O*) if *O* is a resource or (*S,P,O*[*Ts, Te*)) if *O* is a literal. Here *S*, *P,* and *O* are, respectively, the subject, predicate, and object of triple, $Ts, Te \in T$ (*T* is a time domain) and $Ts \leq Te$. The individual terms are described as follows.

- (*S,P,O*) is a common triple of the traditional RDF model.
- When *O* is a resource, *P* may be associated with a timestamp, and *P*[*Ts,Te*) is a temporal predicate of tRDF triple, indicating that the relationship between two resources, *S* and *O,* is valid during the time interval [*Ts,Te*).
- When *O* is a literal, *O* may be associated with a timestamp and *O*[*Ts,Te*) is a temporal literal of the tRDF triple, indicating that the literal *O* is valid during the time interval [*Ts,Te*).
- *T* is a time domain (a set of time points). For $t \in T$, the data type of *t* is *xsd:date* with the format of “yyyy-MM-dd”.
- A timestamp temporal of tRDF triple is represented by a time interval [*Ts,Te*), where $Ts, Te \in T$. As a special case, it is allowed for $Ts = Te$, which signifies a time point instead of a time interval.

Let us look at the tRDF model shown in Table 2. It contains 6 temporal tRDF triples in N-Triples format. These triples describe the personal information of *Márton_Garas*, including *date of birth*, *place of birth*, *name*, *gender*, *date of death*, and *place of death*. They share a common subject, a resource identified by “http://dbpedia.org/resource/Márton_Garas,” and two types of objects. It is shown in Table 2 that, for the object that is a resource, a timestamp in the form of a time interval is attached to the predicate; for the object that is a literal, a timestamp is added to the object. Note that data *date of birth*, *place of birth*, *date of death,* and *place of death* are attached with time points and time intervals with the same start time and the end time.

Table 2 An example of a tRDF model

Subject	Predicate	Object
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/birthDate	"1885–5-18 "^^ < http://www.w3.org/2001/XMLSchema#date >
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/birthPlace[1885-05-18.9999-12-31)	http://dbpedia.org/resource/Novi_Sad
http://dbpedia.org/resource/Márton_Garas	http://xmlns.com/foaf/0.1/name	"Márton Garas[1885–05-18, 1930–06-26)" @en
http://dbpedia.org/resource/Márton_Garas	http://xmlns.com/foaf/0.1/gender	"male[1885–05-18,1930–06-26)"@en
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/deathDate	"1930–6-26" ^^ < http://www.w3.org/2001/XMLSchema#date
http://dbpedia.org/resource/Márton_Garas	http://dbpedia.org/ontology/deathPlace[1930-06-26, 9999-12-31)	http://dbpedia.org/resource/Budapest

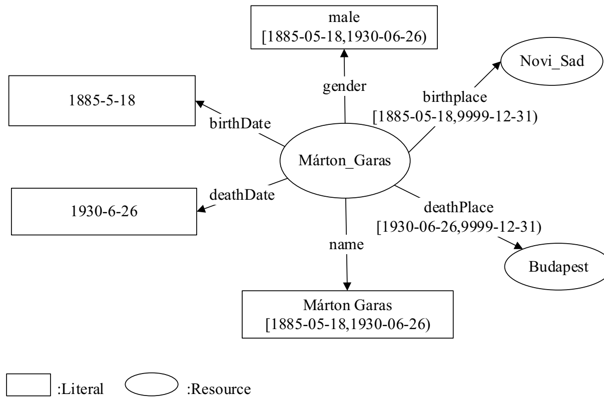

Fig. 2 An example of a tRDF graph

4.2.2 tRDF graph

The tRDF model can be represented as a directed graph. As shown in Fig. 3, in the tRDF graph model, nodes *S* and *O* represent the subject and object of the tRDF triple, respectively. When the object is a resource, the directed edge *P*[*Ts,Te*) represents a temporal predicate of the triple, i.e., a temporal relationship between *S* and *O*. When the object is a literal, the directed edge *P* represents a static predicate of the triple, and meanwhile, the node *O*[*Ts,Te*) represents a temporal object of the triple, which is a temporal value of *S* on *P*.

Note that by deriving temporal information from the tRDF graph representation, a tRDF graph can be converted to an ordinary RDF graph. For this purpose, it is required to introduce several new vocabularies (e.g., startTime and endTime) to describe the temporal interval. In the tRDF graph, for the nodes with temporal interval, we first introduce a new node *T* to represent the time interval and then use the startTime and endTime vocabularies to represent the start and end times of node *T*. As to the subject, predicate, and object of tRDF triples, they are represented with the vocabularies *rdf:subject*, *rdf:predicate,* and

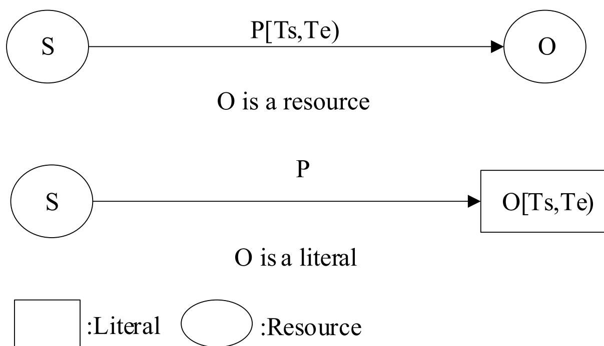

Fig. 3 Graphic representation of tRDF triple

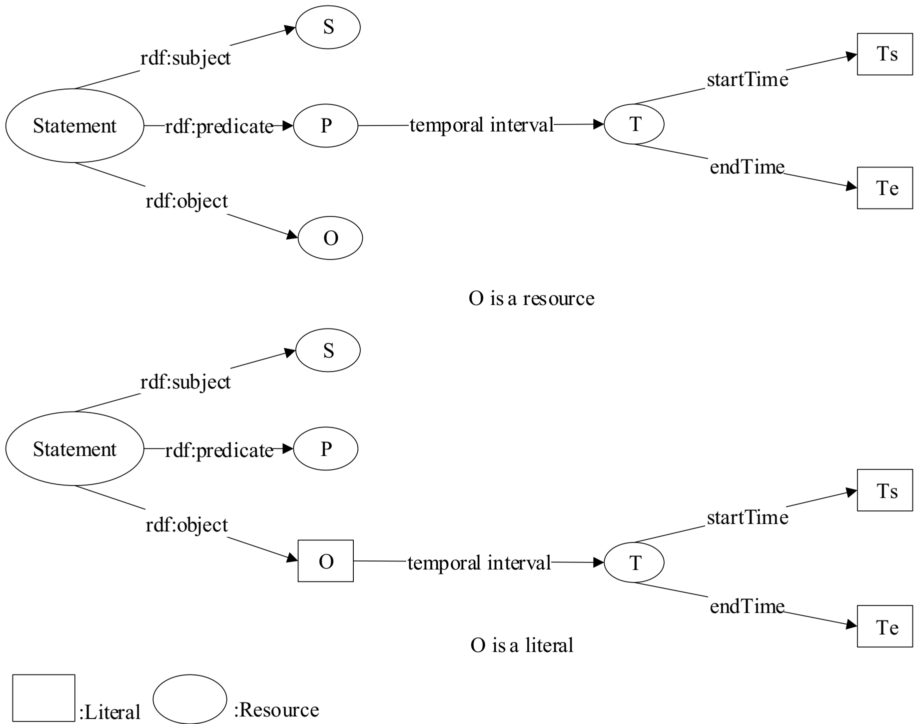

Fig. 4 Ordinary RDF graph converted from the corresponding tRDF graph

rdf:object, respectively. Note that, unlike the existing temporal RDF models, the tRDF model is converted according to the type of object. For two temporal RDF triples in Fig. 3, they are converted to the ordinary RDF graph shown in Fig. 4.

4.3 tRDF semantics

The semantics of the classical RDF model includes three aspects, which are the explanation, satisfaction, and entailment of the RDF model. As for the tRDF model, its semantics are also described from these three aspects.

4.3.1 Temporal interpretation

As with the classical RDF model, the tRDF model is interpreted using expressions or logical relational operators except with added temporal information.

Definition 2 (Temporal interpretation) Let *I* be the interpretation of the RDF model and *TI* be the interpretation of the tRDF model, where the RDF model is obtained by removing all temporal information from the tRDF model. Then *TI* is defined by adding the following temporal elements into *I*:

- A subset T of IR indicates the set of interval information.
- A flag OR indicates that the object is a resource.
- A subset BP of IP–basic properties, indicates the set of predicates without temporal information when the object is a literal.
- A subset TP of IR–temporal properties indicates the set of predicates with temporal information when the object is a resource. Also, temporal-related contents need to be added to TP (e.g., startTime and endTime).
- A subset BO of IR–basic objects, indicating the set of objects without temporal information when the object is a resource.
- The literal set IL needs several temporal properties (e.g., startTime and endTime).
- A mapping PT, mapping $TP \times (T \cap OR) \times (T \cap OR)$ to IP.
- A mapping ILR, mapping $IL \times T \times T$ to IR.

4.3.2 Temporal satisfaction

The satisfaction of the temporal RDF model means the basic semantic relationship between the interpretation TI of the tRDF model and temporal RDF triples.

Definition 3 (Temporal satisfaction) Given an interpretation TI of the tRDF model TM, TI satisfies a certain triple $tm \in T$ (written as $TI \models tm$), if and only if.

- $\forall Ts$, $Te \in T$, $(S,P,O) \in (TI(Ts) \wedge TI(Te))$, we have $TI \models (S,P[Ts,Te),O)$ when O is a resource;
- $\forall Ts$, $Te \in T$, $(S,P,O) \in (TI(Ts) \wedge TI(Te))$, we have $TI \models (S,P,O[Ts,Te))$ when O is a literal.

If $TI \models tm$ for $\forall tm \in TM$, then it can be said that the temporal interpretation TI satisfies the tRDF model TM, written as $TI \models TM$.

4.3.3 Temporal entailment

Temporal entailment represents the logical relationship between two entities (e.g., temporal inclusion), which is mainly used for intellectual reasoning and logical deduction.

Definition 4 (Temporal entailment) Let TM be the tRDF model and TI be an interpretation of TM.

When O is a resource,

- $\forall Ts$, $Te \in T$, $(S,subP,O) \in (TI(Ts) \wedge TI(Te))$, we have $TI \models (S,subP[Ts,Te),O)$;
- If $TI \models (S,P[Ts,Te),O)$ and there exists a time interval $[Ts',Te')$ ($Ts \leq Ts' \leq Te' \leq Te$), there is $TI \models (S,P[Ts',Te'),O)$.

When O is a literal,

- $\forall Ts$, $Te \in T$, $(S,P,subO) \in (TI(Ts) \wedge TI(Te))$, we have $TI \models (S,P,subO[Ts,Te))$;
- If $TI \models (S,P,O[Ts,Te))$ and there exists a time interval $[Ts',Te')$ ($Ts \leq Ts' \leq Te' \leq Te$), there is $TI \models (S,P,O[Ts',Te'))$.

5 Storage of tRDF

In this paper, we store tRDF data with SQL:2011, which supports temporal data manipulation. We first propose the relational schema designed for temporal RDF storing and then propose the rules and algorithms of mapping tRDF data to the relational databases.

5.1 Design of database schema

Among the three methods of storing the classical RDF data with relational databases, the horizontal stores suffer from problems such as multi-valued attributes and many null values (or the horizontal stores with a single table) or too many built tables (for the horizontal stores with multiple tables); the type store is applicable to the scenarios that the RDF triples hold more types of subjects, and may have problems of multi-valued attributes, some null values and some built tables (Ma et al., 2016). Many built tables for RDF triple store mean that many join operations are generally involved for querying. In addition, with the horizontal and type stores, when new triples are inserted, new predicates must result in dynamic relational schema(s).

Based on the above understanding, in this paper, we adopt the basic idea of the vertical stores to store the tRDF data with SQL:2011. To overcome the shortages of the classical vertical stores and satisfy the need to store temporal information, we designed five tables, named the *Namespace table*, *Subject table*, *Property table*, *Object table,* and *Statement table*, respectively, rather than a single table. The schemas of these five relational tables are defined as follows.

Definition 5 (Schema of the relational table) The schema of the relational table is a six-tuple $P=(N, COL, DT, PK, FK, L)$.

(1) $N=TN \cup DN$ is a finite non-empty set of names, where TN is a set of names of the entity tables and DN is a set of names of data type;
(2) COL is a finite non-empty set of column names of the table. For $\forall t \in TN$, we have $\exists COL(t)$;
(3) DT is a set of data types of columns of the table. For $\forall c \in COL(t)$, we have $\exists DT(c) \in DN$;
(4) PK is a set of primary keys of the table. For $\forall t \in TN$, we have $\exists PK(t) \in COL(t)$;
(5) FK is a set of foreign keys of the table. For $\forall t \in TN$, we have $\exists n(n \geq 0)\ FK(t) \subseteq COL(t)$;
(6) $L \subseteq TN \times TN$ is a set of relationships between the tables. The relationships between tables are represented by the reference from the foreign key $FK(t_i)$ to the primary key $PK(t_j)$. For $\forall t_i, t_j \in TN$, a reference to table t_j's primary key $PK(t_j)$ by table t_i's foreign key $FK(t_i)$ can be indicated as $FK(t_i) \rightarrow PK(t_j)$.

The Property table shown in Table 3 contains the *ID*, *NS_ID*, *Property*, *PT*s, and *PT*e columns. The type of *ID* column is *BIGSERIAL*, which means self-increment. *PRIMARY KEY* indicates that the *ID* column is the table's primary key, which can uniquely identify a record. *NOT NULL* means it is not allowed for the column to be empty. The *NS_ID* column is used to represent the *IRI* prefix of the complete predicate

Table 3 Property table creation with SQL:2011

String sqlcreateP = "CREATE TABLE Property(			
ID	BIGSERIAL	PRIMARY KEY	NOT NULL,
NS_ID	INT	REFERENCES NAMESPACE(ID)	NOT NULL,
Property	TEXT		NOT NULL,
PTs	DATE		,
PTe	DATE		,
PERIOD FOR ProPeriod (PTs,PTs))" ;			

Table 4 Namespace table

ID(PK)	Prefix
1	http://dbpedia.org/resource
…	…

stored in the Namespace table. References Namespace (*ID*) keyword means that this column is the table's foreign key, which refers to the *ID* column in the Namespace table. The Property and columns *PT*s and *PT*e respectively correspond to the predicate and time information,. Here it is allowed for the PTs and PTe columns to be empty, and their data types must be the same. As we know, for a tRDF triple, a timestamp is attached to its predicate when its object is a resource; a timestamp is attached to its object when its object is a literal, where *PT*s and *PT*e columns are empty. The *PERIOD FOR* keyword is used to define the valid time, and the *ProPeriod* is the name of the time interval.

The structure of the Namespace table is shown in Table 4. This table, which is applied to store the prefix of the tRDF triple, consists of the primary key *ID* and the Prefix column. There are many duplicate *IRI*s in the tRDF data, and separating the IRI from the subject, predicate, and object can significantly save storage space. The Namespace table stores the IRIs in the Prefix column and is linked to the Subject, Predicate, and Object tables by the primary key *ID*.

The structure of the Subject table is shown in Table 5. This table, which is applied to store the subject of the tRDF triple and associate with the Statement table through the primary key *ID* column, consists of the primary key *ID*, the foreign key *NS_ID,* and the Resource column. The NS_ID column acts as a foreign key to refer to the prefix of the subject stored in the Namespace table. The Resource column stores the subjects without prefixes.

The structure of the Property table is shown in Table 6. This table, which is applied to store the predicate of the tRDF triple and associate the Statement table through the primary key *ID*, consists of the primary key *ID*, the foreign key *NS_ID*, the Property,

Table 5 Subject table

ID(PK)	NS_ID(FK)	Resource
1	1	Fiatau_Penitala_Teo
…	…	…

Table 6 Property table

ID(PK)	NS_ID(FK)	Property	PTs	PTe
1	2	22-rdf-syntax-ns#type	1965–11-07	1996–10-21
…	…	…	…	…

Table 7 Object table

ID(PK)	NS_ID(FK)	Object	OTs	OTs
1	3	Person		
…	…	…	…	…

Table 8 Statement table

ID(PK)	Sid(FK)	Pid(FK)	Oid(FK)
1	1	1	1
…	…	…	…

the *PT*s, and the *PT*e column. The *NS_ID* column acts as a foreign key to refer to the prefix of the predicate stored in the Namespace table. The Property column stores the predicates without prefixes. When the object is a resource, the *PT*s and *PT*e columns are, respectively, the start and end time of the time interval, and the *PT*s and *PT*e columns are empty when the object is a literal.

The structure of the Object table is shown in Table 7. This table, which is applied to store the object of the tRDF triple and associate the Statement table through the primary key *ID*, consists of the primary key *ID*, the foreign key *NS_ID*, the *Object*, the *OT*s, and the *OT*e column. Here the *NS_ID* column acts as a foreign key to refer to the prefix of the object stored in the Namespace table. Note that when the object is a literal, the *NS_ID* of the record corresponds to the *ID* with a null prefix in the Namespace table. The Object column stores the objects without prefixes. When the object is a resource, the *OT*s and *OT*e columns are empty, and the *OT*s and *OT*e columns are, respectively, the start and end time of the time interval when the object is a literal.

The structure of the Statement table is shown in Table 8. This table, which is applied to store the statement of tRDF triple by using integers, consists of the primary key *ID*, the foreign key *Sid*, the foreign key *Pid,* and the foreign key *Oid* column. The Statement table uses foreign keys, *Sid*, *Pid,* and *Oid,* to refer to the subjects, predicates, and objects stored in the tables.

The above tables are connected through their primary keys and foreign keys. The relationships between these tables are shown in Fig. 5.

5.2 Mapping rules

Based on the relational schemas designed in Sect. 5.1, we present the rules for mapping the tRDF model to the relational tables. First, we need to divide the subjects, predicates, and objects of tRDF triples into prefix *N*, subject *ES* without prefix, predicate *EP* without

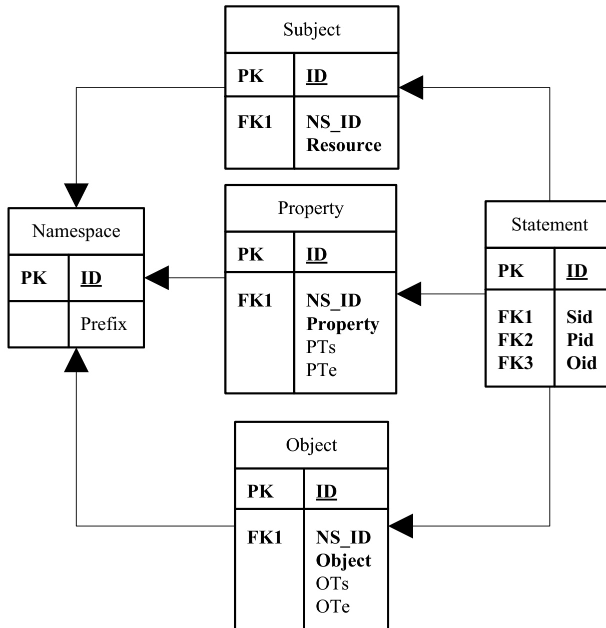

Fig. 5 Relationships of the created relational schemas

prefix, object *EO* without prefix, and temporal information *Ts* and *Te*. On this basis, the mapping rules for each part of tRDF triples are given as follows.

- *Rule 1*: Insert the prefix *N* into the Prefix column of the Namespace table. Note that when the object is a literal, the Prefix column is allowed to be null according to the structure setting of the Namespace table.
- *Rule 2*: Insert the prefix *ID* in the Namespace table returned by *Rule* 1, and then insert the corresponding subject *ES* without prefix into the *NS_ID* and Resource columns of the Subject table, respectively. The *ID* column in the Subject table corresponding to each record acts as the primary key referred to in the Statement table.
- *Rule 3*: Insert the prefix *ID* in the Namespace table returned by *Rule 1* and the corresponding predicate *EP* without prefix into the *NS_ID* and the Property columns of the Property table, respectively. When the object is a resource, insert the temporal information *Ts* and *Te* into the *PT*s and PTe columns of the Property table, respectively. The *ID* column in the Property table corresponding to each record acts as the primary key referred to in the Statement table.
- *Rule 4*: Insert the prefix *ID* in the Namespace table returned by *Rule 1* and the corresponding Object *EO* without prefix into the *NS_ID* and the Object columns of the Object table, respectively. When the object is a literal, insert the temporal information *Ts* and *Te* into the *OT*s and *OT*e columns of the Object table, respectively. The *ID* column in the Object table corresponding to each record acts as the primary key referred to in the Statement table.
- *Rule 5*: When the *NS_ID*, *Property,* and *Object* columns are the same as the situations in *Rules 3* and *4*, the following four cases need to be considered according to different temporal information.

- Discard: if $(PTs \bigvee OTs) \leq Ts \bigwedge (PTe \bigvee OTe) \geq Te$, the data will be discarded.
- Insert: if $(PTs \bigvee OTs) > Te$ or $(PTe \bigvee OTe) < Ts$, the data will be inserted.
- Cover: if $(PTs \bigvee OTs) > Ts \bigwedge (PTe \bigvee OTe) < Te$, the original data in the tables will be covered.
- Combine: if $(PTs \bigvee OTs) > Ts \bigwedge (PTe \bigvee OTe) \geq Te$ or $(PTs \bigvee OTs) \leq Ts \bigwedge (PTe \bigvee OTe) < Te$, the original data in the tables will be combined with the new data.
- Rule 6: Insert the *ID* of the subject, predicate, and object of each tRDF triple returned by *Rules 2, 3,* and *4* into the *Sid*, *Pid,* and *Oid* columns of the Statement table.

5.3 Mapping algorithms

To store tRDF data in relational databases, the original tRDF data need to be analyzed first. Through tRDF data analysis, the prefix, subject, predicate, object, and temporal information of the temporal RDF triples can be obtained. The details of the tRDF data analysis are shown in Algorithm 1.

Algorithm 1 tRDF Data Analysis[7]

Inputs: tRDF triples
Output: Prefix *N*, simple subject *ES*, simple predicate *EP*, simple object *EO*, temporal information *Ts* and *Te*

```
1   while(iter.hasNext()) do
      /* Jena7 API (Application Programming Interface) is first used to obtain the subject, predicate and object of the tRDF triple. */
2     String subject = stmt.getSubject().toString(); // get the subject
3     String predicate = stmt.getPredicate().toString();// get the property
4     RDFNode objectN = stmt.getObject();
5     Srting object = stmt.getObject().toString();// get the object
      /* The predicate and object are then analyzed to obtain the time information according to the type of object. */
      /* When the object is a resource, the temporal information will be obtained from the predicate. */
6     if objectN instanceof Resource
7       String predicate1 = predicate.substring(0, predicate.indexOf("["));
8       String T = predicate.substring(predicate.indexOf("[")+1, predicate.length()-1);
9       String Ts = T.substring(0, T.indexOf(","));//get the Ts
10      String Te = T.substring(T.indexOf(",")+1,T.length());//get the Te
11    end if
      /* When the object is a literal, the operation of obtaining temporal information from the object is similar to the above. */
      /* Finally, the prefix is split from the subject, predicate and object. */
12    String[] S = subject.split("/");
13    String easyS = S[S.length-1];
14    String regexS ="/"+easyS;
15    String namespaceS = subject.substring(0, subject.indexOf(regexS));
      /* The operation of the predicate and object is similar to the above. */
16  end While()
```

With the mapping rules proposed in Sect. 5.2, the algorithm for data storage can store the prefix, subject without prefix, predicate without prefix, object without prefix, and temporal information of the tRDF triples obtained with Algorithm 1 into the relational tables. The details of the tRDF data store are shown in Algorithm 2.

[7] https://jena.apache.org/

Algorithm 2 Storage of tRDF-to-relational table

```
Inputs:   tRDF data after analysis (see Algorithm 1)
Output:   Data in a relational table.
1    while(iter.hasNext()) do
       /* If the prefixes of the subject, predicate, and object are not in the Namespace table, the prefixes will be inserted. */
2      if (isNotExistN(namespaceS))
3        String sqlInsertNamespace = "INSERT INTO namespace (prefix) VALUES (?);";
4        executeUpdate (sqlInsertNamespace);
5      end if
       /* If the current subject is not in the Subject table, the ID corresponding to the prefix and the subject without prefix will be inserted. */
6      if (isNotExistS(S_ID,easyS))
7        String sqlInsertsubject = "INSERT INTO subject (ns_id,resource) VALUES (?,?);";
8        executeUpdate (sqlInsertsubject);
9      end if
       /* When the object is a resource, the temporal information needs to be stored in the Predicate table. */
10     if objectN instanceof Resource
         /* If the current predicate is not in the Property table, the ID corresponding to the prefix, the predicate without prefix, and temporal information
         will be inserted. */
11       if (isNotExistP(P_ID,easyP,ts,te))
12         String sqlInsertproperty = "INSERT INTO property (ns_id,property,pts,pts) VALUES (?,?,?,?);";
13         executeUpdate (sqlInsertproperty);
14       end if
         /* When the predicate without a time interval already exists, the analysis of temporal information is required. */
15       if (isExistP1(P_ID,easyP))
16         if (isDiscard(ts,te)) /* Discard directly, no storage required. */
17         if (isInsert(ts,te))
18           String sqlInsertproperty = "INSERT INTO property (ns_id,property,pts,pts) VALUES (?,?,?,?);";
19           executeUpdate (sqlInsertproperty);
20         end if
21         if (isCover(ts,te)||isCombine(ts,te))
22           modify(P_ID,easyP,ts,te);
23         end if
24       end if
         /* If the current object is not in the Object table, the ID corresponding to the prefix and the object without prefix will be inserted. */
25       if (isNotExistO(O_ID,easyO))
26         String sqlInsertobject = "INSERT INTO object (ns_id,object) VALUES (?,?);";
27         executeUpdate (sqlInsertobject);
28       end if
29     end if
       /* The operation is similar to the above when the object is a literal. */
       /* If the current tRDF triple is not in the Statement table, the IDs corresponding to the subject, predicate, and object of the triple will be inserted. */
30     if(isNotExistStm(Sid,Pid,Oid))
31       String sqlInsertstatement = "INSERT INTO statement (sid,pid,oid) VALUES (?,?,?);";
32       executeUpdate (sqlInsertstatement);
33     end if
34   end While()
```

6 Query of Temporal RDF

For the temporal RDF model tRDF, the traditional RDF query language SPARQL should be extended to support tRDF data query. In this paper, we propose such a query language for the tRDF, termed as tSPARQLt. In this section, we first describe tSPARQLt from two aspects of syntax and basic query statement. Furthermore, to query the tRDF data stored in the temporal relational databases with tSPARQLt, it is necessary to transform the tSPARQLt queries into the corresponding SQL queries.

6.1 Syntax of tSPARQLt

Similar to the SPARQL syntax, we present the tSPARQLt syntax with terms and triple. In this paper, we use the N-Triples format to represent tRDF triples.

6.1.1 tRDF terms

Following are the four basic terms of the tSPARQLt syntax.

- The syntax for *IRI*s. The subject, predicate, and object with a resource type of each RDF triple in SPARQL are composed of complete *IRI*s, where " < " and " > " are delimiters and not part of the *IRI* reference. In the tRDF model, a timestamp is attached to the predicate of a triple when its object is a resource. As a result, in tSPARQLt, when the object is a resource, a time interval [*Ts,Te*) may be attached after the predicate *IRI* of the tRDF triple.
- The syntax for literals. Literals are used to represent string, date, number, or Boolean in SPARQL. A string literal is surrounded by quotation marks, which is followed by the language type quoted by "@" or the IRI identifier that indicates the string type quoted by "^^." A date literal is similar to the string type, where the date is represented as a string but followed by an IRI identifier that indicates the date type. A number literal (e.g., *INTEGER*, *DECIMAL,* and *DOUBLE*) is interpreted as the numeric meaning of the corresponding type, which does not have any quotation marks or is not followed by an IRI to specify the data type. A Boolean literal can be written directly as TRUE or FALSE. In the tRDF model, a timestamp may be attached to the object that is a literal. So, for the tRDF triple whose object is a literal, in tSPARQLt, the time interval [*Ts,Te*) should be added after the literal object of the tRDF triple.
- The syntax for query variables. SPARQL identifies a query variable by prefixing it with mark "?" or "$," but they are not part of the variable name. SPARQL has three types of query variables: *?S*, *?P,* and *?O*, which denote the query on the subject, predicate, and object of RDF triples, respectively. For the tRDF model, two new query variables *?Ts* and *?Te* are introduced to tSPARQLt, which are used to represent the query on the start and end times of time intervals, respectively. Then tSPARQLt can query the variables *?Ts* and *?Te* on the predicate when the object is a resource, and query the variables *?Ts* and *?Te* on the object when the object is a literal.
- The syntax for blank nodes. The blank nodes in tSPARQLt are consistent with SPARQL.

6.1.2 tRDF triple pattern

A triple pattern in SPARQL is a special triple, and at least one of its subject, predicate, or object is a query variable, for example, {*?S ?P ?O.*}. In the tRDF model, a temporal triple contains temporal information on its predicate or object. Correspondingly, a triple pattern in tSPARQLt should have two new variables *?Ts* and *?Te* to represent time information. We identify two types of tRDF triple patterns as follows:

- {*?S ?P*[*?Ts,?Te*) *?O.*}) for the case that the object is a resource.
- {*?S ?P ?O*[*?Ts,?Te*)*.*} for the case that the object is a literal.

6.2 Query statement of tSPARQLt

The query statement structure of tSPARQLt is similar to SPARQL, which contains four clauses and is summarized in Table 9. These four clauses correspond to the query form,

Table 9 tSPARQLt query statement

```
SELECT [DISTINCT] * | SELECT [DISTINCT] ?variable name1 ?variable name2…
[FROM  tRDF Dataset | FROM NAMED tRDF Dataset]
WHERE{ { triple pattern1 .
          triple pattern2 .
          [FILTER (conditions1)] }
        { triple pattern3 .
          triple pattern4 .
          [FILTER (conditions2)] }
        OPTIONAL { triple pattern5 .
                   triple pattern6 .
                   [FILTER (conditions3)] }
         …
       }
[ORDER BY|LIMIT|SKIP]
```

dataset, graph pattern with constraints, and solution modifier. In the following, we explain these four clauses in detail.

6.2.1 Query form of tSPARQLt

Like SPARQL, tSPARQLt also includes four query types identified by the keywords *SELECT*, *CONSTRUCT*, *ASK* and *DESCRIBE*, respectively. *SELECT* returns all variables or a subset of the variables that are obtained by a query pattern match. *CONSTRUCT* produces a tRDF graph that is made up of the matched triples. *ASK* determines whether the querying dataset contains the desired match, which returns true or false as a result. *DESCRIBE* returns tRDF information for all resources that match the query. Among these four types of queries, *SELECT* is very useful for data retrieval.

The use of SELECT of tSPARQLt is basically the same as the SELECT of SPARQL. First, the statement *SELECT ** returns all variables and their values bound to these variables. Second, the statement *SELECT ?variable1 ?variable2…* only returns the variables and their bindings that are specified by the given variable names. The main difference between the SELECT of tSPARQLt and the SELECT of SPARQL is that they use different RDF models. The tRDF model contains temporal information, and the SELECT of tSPARQLt may contain two query variables *?Ts* and *?Te*, which describes the start time and end time of a time interval. So, the SELECT of tSPARQLt can query the time variables *?Ts* and *?Te* and the SELECT of SPARQL cannot.

6.2.2 Dataset in tSPARQLt

The dataset in tSPARQLt all means the data resource identified by the URL enclosed by "[]." The dataset can include zero or more named graphs. Like SPARQL, tSPARQLt always contains a default graph. A query statement without the clause *FROM* means that the query is evaluated against a default graph. When a named graph is specified in the query statement, which means a match to the named graph only, both the clauses *FROM* and *FROM NAMED* can be used in the query. The former is for a traditional RDF dataset,

and the latter is for a temporally extended tRDF dataset. The specific form of the dataset in tSPARQLt is presented in Table 10.

6.2.3 Graph patterns of tSPARQLt

Like SPARQL, the queries of tSPARQLt are evaluated based on graph pattern matching. Nevertheless, the graph pattern of tSPARQLt is different from the graph pattern of SPARQL because tSPARQLt is for the tRDF model, and its graph pattern should contain temporal information also.

The statement *SELECT ?Ts ?Te WHERE* {*?S P*[*?Ts,?Te*) *?O*}, for example, matches all triples with the given predicate *P*, regardless of the subject, object, and temporal information. For the matched triples whose objects are a resource, the start time and end time of their predicates can be returned by the statement with pattern {*?S P*[*?Ts,?Te*) *?O*}. And the statement *SELECT ?Ts ?Te WHERE* {*?S ?P O*[*?Ts,?Te*)} matches all triples with the given object *O*, regardless of the subject, predicate, and temporal information. For the matched triples whose objects are a literal, the start time and end time of their objects can be returned by the statement with pattern {*?S ?P O*[*?Ts,?Te*)}. Also, the statement *SELECT ?S WHERE* {*?S P*[*Ts,Te*) *?O*} matches all triples with the given predicate *P*[*Ts,Te*). This statement with pattern {*?S P*[*Ts,Te*) *?O*} can return subjects of the matched triples whose objects are a resource. Furthermore, the statement *SELECT ?S WHERE* {*?S ?P O*[*Ts,Te*)} matches all triples with the given object *O*[*Ts,Te*). This statement with pattern {*?S ?P O*[*Ts,Te*)} can return subjects of the matched triples whose objects are a literal.

The graph pattern of tSPARQLt can combinedly use three conventional variables (i.e., *?S*, *?P* and *?O*) and two temporal variables (i.e.,*?Ts* and *?Te*). With the above examples of basic graph patterns, we can construct diverse graph patterns for tSPARQLt, including *basic graph patterns*, *group graph patterns,* and *optional graph patterns*.

Basic graph pattern tSPARQLt uses graph patterns to match and filter tRDF data, and graph patterns are, therefore, the most important part of tSPARQLt query statements. A graph pattern is built with the basic graph patterns (BGPs), and a basic graph pattern consists of multiple triple patterns. For where all triple patterns need to be matched. Of course, it is possible that a basic graph pattern only contains a triple pattern. SPARQL contains eight basic triple patterns, depending on the combination of different query variables. In tSPARQLt, the basic triple patterns are greatly extended by adding two new query variables about time information. Table 11 presents the basic triple patterns in tSPARQLt when the object is a resource. In addition, there are similar basic triple patterns in tSPARQLt when the object is a literal, which are not shown here.

Group graph pattern The group graph pattern in tSPARQLt consists of a set of basic graph patterns, in which each basic graph pattern must be matched for query evaluation.

Table 10 Dataset declaration in tSPARQLt

```
SELECT *
FROM <URL1>
FROM NAMED <URL2>
WHERE{Graph Pattern.}
```

Table 11 Triple patterns of tSPARQLt when the object is a resource

?S P[Ts,Te) O	*S ?P[Ts,Te) O*	*S P[?Ts,Te) O*
S P[Ts,?Te) O	*S P[Ts,Te) ?O*	*?S ?P[Ts,Te) O*
?S P[?Ts,Te) O	*?S P[Ts,?Te) O*	*?S P[Ts,Te) ?O*
S ?P[?Ts,Te) O	*S ?P[Ts,?Te) O*	*S ?P[Ts,Te) ?O*
S P[?Ts,?Te) O	*S P[?Ts,Te) ?O*	*S P[Ts,?Te) ?O*
?S ?P[?Ts,Te) O	*?S ?P[Ts,?Te) O*	*?S ?P[Ts,Te) ?O*
?S P[?Ts,?Te) O	*?S P[?Ts,Te) ?O*	*?S P[Ts,?Te) ?O*
S ?P[?Ts,?Te) O	*S ?P[Ts,?Te) ?O*	*S P[?Ts,?Te) ?O*
?S ?P[?Ts,?Te) O	*?S ?P[?Ts,Te) ?O*	*?S P[?Ts,?Te) ?O*
S ?P[?Ts,?Te) ?O	*?S ?P[?Ts,?Te) ?O*	*S P[Ts,Te) O*

In tSPARQLt, the basic graph pattern is composed of triple patterns. Let us look at an example *SELECT ?S ?O WHERE* {*?S P1*[*Ts1,Te1*) *?O. ?S P2*[*Ts2,Te2*) *?O. ?S P3*[*Ts3,Te3*) *?O.*}. This statement matches all triples whose predicates are *P1*[*Ts1,Te1*), *P2*[*Ts2,Te2*) and *P3*[*Ts3,Te3*), which returns the subjects and objects of such triples.

Optional graph pattern The optional graph pattern in tSPARQLt, which is identified by the keyword *OPTIONAL*, contains basic graph patterns or group graph patterns that may not be matched in query evaluation. In tSPARQLt, the triple patterns are used to construct the optional graph pattern. The statement *SELECT ?S ?O WHERE* {*?S P1*[*Ts1,Te1*) *?O. OPTIONAL* {*?S P2*[*Ts2,Te2*) *?O*}}, for example, matches all triples whose predicates must be *P1*[*Ts1,Te1*) and may be *P2*[*Ts2,Te2*), which returns the subjects and objects of such triples.

Constraints in graph pattern Like SPARQL, tSPARQLt also uses the keyword *FILTER* to filter the results obtained with the graph patterns. The clause *FILTER* can only occur in the graph patterns. There are two major scenarios to use *FILTER*: restricting the value of strings and restricting the values of some types (e.g., numeric types, xsd:string, xsd:boolean and xsd:dateTime). We discuss the *FILTER* of these two scenarios in tSPARQLt as follows.

- Restriction of string. The *FILTER* of tSPARQLt also uses the function *regex*() to match string literals. Using the str function, *regex*() can also match the lexical forms of other literals. The clause *WHERE* {*?S P*[*Ts,Te*) *?O. FILTER REGEX* (*?O, "^Knowledge Graphs", "i"*)}, for example, searches all triples whose predicates are *P*[*Ts,Te*) and whose objects contain *"Knowledge Graphs."* Here the "*i*" flag indicates that the match in *regex*() is case-insensitive. The *FILTER* of tSPARQLt is not different from the *FILTER* of SPARQL in this scenario.
- Restriction of some data types. The *FILTER* can be used to restrict expressions, which consist of variables, operators, and RDF terms. In tSPARQLt, the used variables can be ?*Ts* and ?*Te*, and the tRDF terms can be time values. The clause *WHERE* {*?S P*[*?Ts,?Te*) *?O. FILTER* (*?Ts > 2007–01-02*)}, for example, can filter the matched triples by setting the start time of the predicate on January 2, 2007.

To simplify time expressions in the *FILTER* of tSPARQLt, it is necessary to use temporal predicates. Temporal query languages have been widely studied in the context of relational databases. Following SQL:2011, we introduce and apply seven temporal

predicates in the *FILTER* of tSPARQLt, including *CONTAINS*, *OVERLAPS*, *EQUALS*, *PRECEDES*, *SUCCEEDS*, *IMMEDIATELY PRECEDES*, and *IMMEDIATELY SUCCEEDS*. Their usage samples are as follows.

(1) *FILTER* (X *CONTAINS* Y)
(2) *FILTER* (X *OVERLAPS* Y)
(3) *FILTER* (X *EQUALS* Y)
(4) *FILTER* (X *PRECEDES* Y)
(5) *FILTER* (X *IMMEDIATELY PRECEDES* Y)
(6) *FILTER* (X *SUCCEEDS* Y)
(7) *FILTER* (X *IMMEDIATELY SUCCEEDS* Y)

Here, X can be a temporal variable such as *?Ts* and *?Te*, or a pair like (*?Ts*, *?Te*); Y can be a time point such as 2020–1-1, or a time period such as (2020–1-1, 2021–12-31).

6.2.4 Solution modifiers of tSPARQLt

The returned results of the tSPARQLt query may be large, unordered, and redundant. Like SPARQL, tSPARQLt can use six modifiers, which are *ORDER BY*, *DISTINCT*, *LIMIT*, *PROJECT*, *REDUCED,* and *OFFSET*, to optimize the result set. With these modifiers, a more intuitive and easy-to-understand sequence is created. The solution modifiers that are commonly used in tSPARQLt are briefly presented as follows.

- *ORDER BY*: It is followed by variable name(s) and an optional order modifier ASC (ascending order) or DESC (descending order), ranking the results in ascending/descending order according to the variable name(s). Here is an example *SELECT ?S WHERE* {*?S P*[*Ts,Te*) *?O.*} *ORDER BY DESC* (*?S*). Noe that tSPARQLt may use two new variables for time, so it can order the results according to the time variables, for example, *SELECT ?S WHERE* {*?S P*[*?Ts,?Te*) *?O.*} *ORDER BY ?Ts*.
- *DISTINCT*: It is followed by a variable name(s) in the statement *SELECT*, eliminating possible duplicates of the result set on the variable name(s), for example, *SELECT DISTINCT ?S WHERE {?S P*[*Ts,Te*) *?O.}*.
- *LIMIT*: It is followed by an integer, indicating the maximum number of returned results, for example, *SELECT ?S WHERE* {*?S P*[*Ts,Te*) *?O.*} *LIMIT 20.*
- *OFFSET*: It is followed by an integer, setting the offset of the returned result. It is often used in conjunction with *LIMIT*, for example, *SELECT ?S WHERE* {*?S P*[*Ts,Te*) *?O.*} *LIMIT 20 OFFSET 10.*

6.3 Transformation from tSPARQLt to SQL

tSPARQLt queries are not supported by relational databases. So, to query tRDF data stored in relational tables with tSPARQLt queries, it is necessary to transform the tSPARQLt queries into the corresponding SQL queries. In this section, we first discuss the rules of transforming the basic query statement in tSPARQLt to SQL. We further present concrete transformation cases where major tSPARQLt queries are transformed into their corresponding SQL statements. With the transformation of basic statements, more complex statements can be transformed.

6.3.1 Transformation rules

The basic query statement of tSPARQLt proposed in Sect. 6.2 contains four clauses. We investigate how to use these four clauses in tSPARQLt.

6.3.2 Transformation of query types

As we know, tSPARQLt includes four types of queries: *SELECT*, *ASK*, *CONSTRUCT* and *DESCRIBE*. Here we only focus on *SELECT*, which is widely applied for RDF data query. The keyword *SELECT* in tSPARQLt directly corresponds to the keyword *SELECT* in SQL. The keyword SELECT of tSPARQLt is followed by variable names prefixed with "?", which are separated by spaces. The keyword SELECT of SQL is followed by column names of tables, which are separated by ",".

First, the statement *SELECT* * is used both in tSPARQLt and SQL. This statement in tSPARQLt returns all variable names and their binding values, and it in SQL returns all records (i.e., tuples) in the entire table. The transformation rules for the statement *SELECT* are summarized as follows.

- The statement *SELECT* * in tSPARQLt can make a direct transformation without any change.
- The statement *SELECT ?Variablename1 ?Variablename2 …?VariablenameN* in tSPARQLt needs to be transformed to the column names (i.e., attributes) of the corresponding tables in SQL, where the columns are named according to the actual meaning of the variable names (i.e., subject, predicate, object or time). The result of the SQL statement looks like *SELECT column1,column2,…, columnN*.

Transformation of the dataset The keyword *FROM* in tSPARQLt directly corresponds to the keyword *FROM* in SQL. The keyword *FROM* of tSPARQLt is followed by the tRDF dataset to be queried, which is identified by the IRI enclosed in "[]." This dataset can be either a default or a named graph that may contain zero or more graphs.

The clause *FROM* can be left out in the tSPARQLt, which means that the query statement can be executed in the default graph. Unlike tSPARQLt, in relational databases, the clause *FROM* must exist and is used to specify the table(s) that contain(s) the column names provided by the clause WHERE. For the transformation of the clause *FROM* in tSPARQLt, it is necessary to determine the tables being used by SQL according to the variables that arise in the clause *SELECT* in tSPARQLt. The transformation rules for the dataset are summarized as follows.

- The query statements in the SQL must contain the statement *FROM*, no matter if there is the statement *FROM* in the tSPARQLt.
- The corresponding relational tables are selected according to the query variables provided in the clause SELECT of tSPARQLt. The selected table names are added to the clause of SQL (e.g., *SELECT column1,column2,…,columnN FROM table_name*).

Transformation of Graph Patterns In tSPARQLt, a graph pattern is introduced by the keyword *WHERE*, which is contained in "{}." For example, the clause *WHERE* {*?S*

P[*Ts,Te*) *?O*} uses a basic graph pattern for the tRDF model, where the objects of triples are a resource. The clause *WHERE* in tSPARQLt directly corresponds to the clause *WHERE* in SQL. Unlike tSPARQLt, the clause *WHERE* in SQL is followed by the query conditions rather than the graph patterns in tSPARQLt. The query conditions of SQL do not need to be enclosed with "{}" and are connected with the keywords AND/OR. The query conditions in SQL must correspond to all triple patterns in the tSPARQLt graph patterns. The key to transforming the graph patterns of tSPARQLt to the query conditions of SQL is to transform the triple patterns contained in the graph patterns.

To transform the clause *WHERE* of tSPARQLt, it is necessary to understand the meaning of each triple pattern in tSPARQLt first. A basic graph pattern {*?S P*[*Ts,Te*) *?O*}, for example, means that it will match all tRDF triples whose objects are a resource and predicates are *P*[*Ts,Te*). The corresponding meaning of this basic graph pattern in SQL is to get all rows whose predicates are *P*, starting and ending at *Ts* and *Te*, respectively. Based on this understanding, the above basic graph pattern is transformed to a clause of SQL *WHERE Predicate* = *P AND PTs* = *Ts AND PTe* = *Te*. In addition, a pluralistic basic graph pattern in tSPARQLt consists of a set of basic graph patterns. For its transformation, each basic graph pattern is first transformed separately, and then these transformed conditions are connected with the keyword OR in SQL.

Moreover, the group graph pattern is a combination of many basic graph patterns. Its transformation is similar to that of the pluralistic basic graph pattern, except for the keyword AND rather than OR. We summarize the transformations of graph patterns in tSPARQLt as follows.

- The keyword *WHERE* in tSPARQLt can be transformed to the keyword *WHERE* in SQL directly.
- In SQL, the "{}" in tSPARQLt is not required, and the keyword WHERE is directly followed by the query conditions.
- A basic graph pattern in tSPARQLt is transformed to the query conditions in SQL. The triple pattern {*S ?P*[*Ts,Te*) *?O*}, for example, corresponds to condition *PTs* = *Ts AND PTe* = *Te* in SQL; the triple pattern {*S ?P*[*?Ts,?Te*) *?O*} corresponds to condition *Subject* = *S* in SQL.
- The transformations of the pluralistic basic graph pattern and group graph pattern in tSPARQLt are on the single basic graph pattern. First, their triple patterns are respectively transformed to the corresponding query conditions in SQL. Then these transformed conditions are connected with AND (for the transformations of pluralistic basic graph pattern) or OR (for the transformations of group graph pattern).

As to the clause *FILTER* used in the graph patterns of tSPARQLt, it is transformed to SQL by adding conditions with the keyword AND. Since the clause *FILTER* only takes effect in the basic graph pattern. For a group graph pattern, the clause *FILTER* in each basic graph pattern is separately transformed into a condition, which should be placed in the query statement of SQL. These conditions are finally connected with the keyword AND. We summarize the transformations of constraints in tSPARQLt as follows.

- A restriction on string values in tSPARQLt is transformed to the corresponding regular expression with the same semantics in SQL.
- A restriction on number or date in tSPARQLt is transformed with the comparison operators and keywords such as NOT, LIKE, IN, NOT IN, BETWEEN in SQL.

- For multiple restrictions in the clause *FILTER* of tSPARQLt, each restriction is first transformed to a condition in SQL with the two transformations above. Then these transformed conditions are connected together with AND. The final conditions are placed in the clause *WHERE* of SQL.

Transformation of solution modifiers Generally speaking, tSPARQLt and SQL use very similar solution modifiers. In the following, we briefly describe some of the common ones. First, the clause *ORDER BY* is used both in tSPARQLt and SQL, which follows the clause *WHERE*. However, the clause ORDER BY is followed by the variable names in tSPARQLt and the column names of the tables in SQL, respectively. In addition,, although tSPARQLt and SQL both use the keywords ASC and DESC, the clauses ORDER BY in tSPARQLt and SQL are applied differently to represent ascending/descending order. In tSPARQLt, the keyword ASC or DESC is followed by a ranking variable, which is enclosed in "()"; In SQL, the keyword ASC or DESC is directly followed by a column name without "()." We summarize the transformations of the clause *ORDER BY* in tSPARQLt as follows.

- Essentially, the variable names in tSPARQLt need to be transformed to the corresponding column names in SQL, maintaining their relative positions.
- *ASC(?variablename)* and *DESC(?variablename)* in tSPARQLt are transformed as *columnname ASC* and *columnname DESC* in SQL, respectively.

Second, the clause DISTINCT is used in the same way both for tSPARQLt and SQL, which follows the keyword *SELECT* and can eliminate duplicate information in the returned results. In addition, the clauses LIMIT and OFFSET are also used in the same way both in tSPARQLt and SQL. The clause LIMIT is applied to limit the number of returned results, and the clause OFFSET is applied to specify the offset of returned results. These two modifiers follow the clause *WHERE* in tSPARQLt and SQL. So, the transformations of the three modifiers from tSPARQLt to SQL are straightforward.

6.3.3 Transformation examples

In the above, we present the principles of transforming tSPARQLt to SQL. In this section, we take the tRDF model whose triple objects are a resource and triple predicates may be time-aware as an example to show how diverse tSPARQLt queries with SELECT are transformed to the corresponding SQL queries. These transformation examples are presented in Table 12. Also, we can have similar transformation examples for querying the tRDF model whose triple objects are a literal and may be time-aware and do not give them due to the limited space.

Note that Table 12 gives some transformation examples for tSPARQLt query statements. With them, we can transform more complex tSPARQLt query statements. For example, we can modify the triple pattern in Example 2 to obtain the Predicate-Object lists, Subject-Predicate lists, and Object lists in the basic graph pattern. Other graph patterns, such as the group graph pattern, can be obtained by extending Example 5, and filtering data can be achieved by adding more constraints on the basis of Example 7. It should be noted that the above transformation examples are only at a conceptual level. In practice, the transformed SQL statements should be optimized against the database structure.

Table 12 Transformation examples of SELECT query for the tRDF model with temporal predicates

<table>
<tr><th></th><th>tSARPQLt</th><th>SQL</th></tr>
<tr><td>1</td><td>SELECT *|?S|?P|?Ts|?Te|?O
[FROM…]
WHERE {S P[Ts,Te) O.}</td><td>SELECT *|SUBJECT|PROPERTY|PTS|PTE|OBJECT
FROM TABLE_NAME
WHERE SUBJECT = S AND PROPERTY = P AND PTS = Ts
AND PTE = Te AND OBJECT = O</td></tr>
<tr><td>2</td><td>SELECT *|?S|?O
[FROM…]
WHERE {?S P[Ts,Te) ?O.}</td><td>SELECT *|SUBJECT|OBJECT
FROM TABLE_NAME
WHERE PROPERTY = P AND PTS = Ts AND PTE = Te</td></tr>
<tr><td>3</td><td>SELECT *|?S|?P|?O
[FROM…]
WHERE {?S ?P[Ts,Te) ?O.}</td><td>SELECT *|SUBJECT|PROPERTY|OBJECT
FROM TABLE_NAME
WHERE PTS = Ts AND PTE = Te</td></tr>
<tr><td>4</td><td>SELECT *|?S|?Ts|?Te|?O
[FROM…]
WHERE {?S P[?Ts,?Te) ?O.}</td><td>SELECT *|SUBJECT|PTS|PTE|OBJECT
FROM TABLE_NAME
WHERE PROPERTY = P</td></tr>
<tr><td>5</td><td>SELECT *|?S|?O
[FROM…]
WHERE {?S P1[Ts1,Te1) ?O
?S P2[Ts2,Te2) ?O
…}</td><td>SELECT *|SUBJECT|OBJECT
FROM TABLE_NAME
WHERE PROPERTY = P1 AND PTS = Ts1 AND PTE = Te1
OR PROPERTY = P2 AND PTS = Ts2 AND PTE = Te2
OR…</td></tr>
<tr><td>6</td><td>SELECT *|?S|?O
[FROM…]
WHERE { ?S P[Ts,Te) ?O
FILTER regex(?S, "hello")
FILTER regex(?O, "world")}</td><td>SELECT *|SUBJECT|OBJECT
FROM TABLE_NAME
WHERE PROPERTY = P ANDP TS = Ts AND PTE = Te
AND SUBJECT ~ 'hello'
AND OBJECT ~ 'world'</td></tr>
<tr><td>7</td><td>SELECT *|?S|?Ts|?Te|?O
[FROM…]
WHERE {?S P[?Ts,?Te) ?O
FILTER(?Ts < '2000–01-01'^^xsd:date)
FILTER(?Te > '2010–01-01'^^xsd:date) FILTER(?Te < '2020–01-01'^^xsd:date)
FILTER…}</td><td>SELECT *|SUBJECT|PTS|PTE|OBJECT
FROM TABLE_NAME
WHERE PROPERTY = P AND PTS < '2000–01-01'
AND PTE BETWEEN '2010–01-01' AND '2020–01-01'
AND …

SELECT *|SUBJECT|PTS|PTE|OBJECT
FROM TABLE_NAME
WHERE PROPERTY = P
AND PPeriod SUCCEED DATE '2000–01-01'
AND PTE BETWEEN '2010–01-01' AND '2020–01-01'
AND …</td></tr>
<tr><td>8</td><td>SELECT DISTINCT ?S
[FROM…]
WHERE {?S P[Ts,Te) ?O.}
ORDER BY DESC(Ts)
LIMIT 30
OFFEST 2</td><td>SELECT DISTINCT SUBJECT
FROM TABLE_NAME
WHERE PROPERTY = P AND PTS = Ts AND PTE = Te;
ORDER BY PTS DESC
LIMIT 30
OFFEST 2</td></tr>
</table>

Discussions. Generally speaking, both SPARQL for RDF and SQL for RDBMS are structure-based query languages with *SELECT-FROM-WHERE*. They have direct correspondences: the clauses *SELECT*, *FROM,* and *WHERE* in SPARQL directly correspond to the clauses *SELECT*, *FROM,* and *WHERE* in SQL, respectively. Also, they use very similar solution modifiers: the clauses *ORDER BY*, *DISTINCT*, *LIMIT,* and *OFFSET* are used in the same way both for SPARQL and SQL. It is demonstrated in Chebotko et al. (2009) that SPARQL-to-SQL translation is semantics preserving. The temporal RDF query

language tSPARQLt and the temporal RDBMS query language SQL:2011 still have direct correspondences and similar solution modifiers. On this basis, the semantics preserving tSPARQLt-to-SQL:2011 translation is completed for all possible situations: *triple patterns*, *basic graph patterns*, *optional graph patterns*, *alternative graph patterns*, and *value constraints*. The core of the tSPARQLt-to-SQL:2011 translation is the clause *WHERE* mapping. Any tSPARQLt queries, including complex ones, constitute one of the above primary situations or a combination of them, and their translations to SQL:2011 are, therefore, semantics preserving. The transformation examples in Table 12 demonstrate the preservation of semantics.

7 Experimental Evaluations

We designed several comparative experiments to verify the feasibility and effectiveness of our proposed storage and query methods in terms of storage time and query efficiency. The experiments were implemented based on the Eclipse platform with JDK13 and PostgreSQL[8]and performed on a system with an Intel(R) Core(TM) i5-4210H 2.9 GHz processor, 8.00 GB RAM, and Windows 10 operating system. Here PostgreSQL is the most powerful open-source relational database management system (RDBMS). Although there are other popular RDBMSs available such as Oracle, MySQL, and SQL Server,[9] PostgreSQL conforms to primary features for SQL:2011 Core conformance. So far, no RDBMS meets full conformance with this standard. To avoid possible errors that may occur in a single experiment, our experimental results given below were obtained by averaging the results of five times of experiments.

7.1 Datasets

There are many classical RDF datasets (Ma et al., 2016). Among them, DBpedia[10] contains 10,310,048 triples describing a multitude of personal information. As shown in Table 1, these triples contain temporal information that appears as elements of triples. Following our temporal RDF model, we re-formatted the triples with temporal information in DBpedia. Basically, we extracted and identified two categories of temporal information and respectively added them as annotations to predicates or objects of the original triples according to the object type. This way, we obtained the tRDF triples, as shown in Table 2. Unlike the temporal RDF dataset, whose temporal information is randomly generated, our tRDF dataset directly comes from the original RDF data with temporal information and is, therefore more authentic and significant. To test the performance of our proposed storage and query methods on datasets with different sizes, we divided the constructed tRDF dataset into four tRDF datasets (i.e., Dataset1, Dataset2, Dataset3, and Dataset4) with different sizes. These four temporal RDF databases are shown in Table 13.

[8] https://www.postgresql.org/

[9] https://db-engines.com/en/ranking

[10] http://wiki.dbpedia.org/about

Table 13 Four temporal RDF datasets with different sizes

Dataset	Number of tRDF triples	Size of dataset (M)
Dataset1	102,606	13.7
Dataset2	1,145,487	153
Dataset3	5,332,453	712
Dataset4	10,310,048	1372

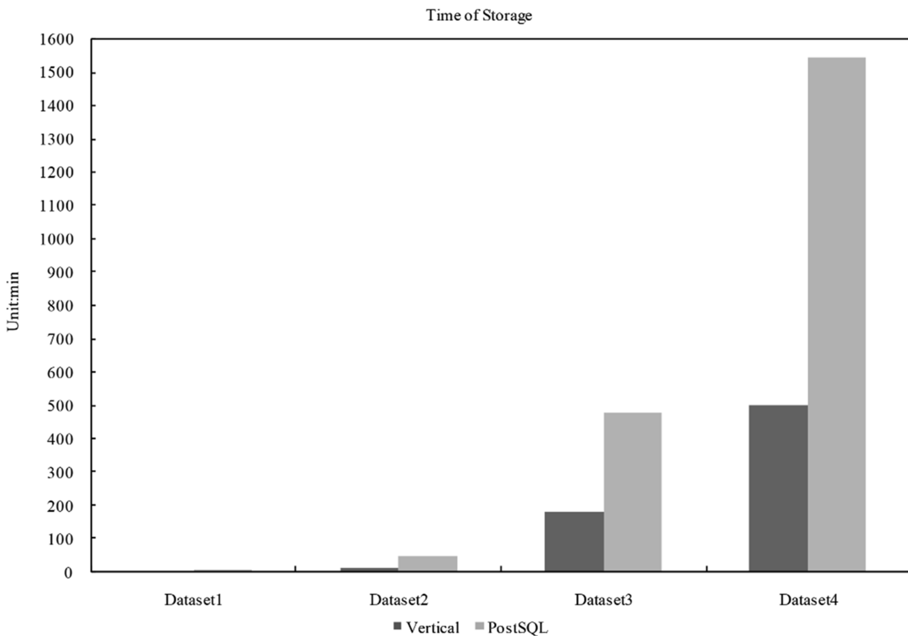

Fig. 6 Time of storage

7.2 Experimental results

We verified the feasibility of tRDF data storage and query methods proposed in the paper from two aspects: storage time and query efficiency. To facilitate the following discussion, our storage method proposed in Sect. 5 is referred to the PostSQL mapping. We compare the PostSQL mapping with the vertical mapping.

7.2.1 Storage time

Each tRDF dataset listed in Table 13 is stored in PostgreSQL with vertical mapping and PostSQL mapping, respectively. The times of storing these four tRDF datasets with these two methods are shown in Fig. 6. It is shown in Fig. 6 that when the number of tRDF triples in the datasets is less than one million, the storage times of both methods increase linearly with the number of triples. However, the storage times increase exponentially with the number of triples when the amount of data exceeds one million.

Table 14 Five query statements

Q1	SELECT * FROM STATEMENT LIMIT 500
Q2	SELECT * FROM STATEMENT WHERE Sid = (SELECT ID FROM SUBJECT WHERE RESOURCE = 'Fiatau_Penitala_Teo' AND ns_id = (SELECT ID FROM NAMESPACE WHERE PREFIX = 'http://dbpedia.org/resource'))
Q3	SELECT * FROM STATEMENT WHERE Sid = (SELECT ID FROM SUBJECT WHERE RESOURCE = 'Fiatau_Penitala_Teo' AND ns_id = (SELECT ID FROM NAMESPACE WHERE PREFIX = 'http://dbpedia.org/resource')) AND Pid = (SELECT ID FROM PROPERTY WHERE PROPERTY = 'gender' AND ns_id = (SELECT ID FROM NAMESPACE WHERE PRE-FIX = 'http://xmlns.com/foaf/0.1'))
Q4	SELECT * FROM STATEMENT WHERE Oid in (SELECT ID FROM OBJECT WHERE TS = '1911–07-23' AND TE BETWEEN '1920–11-01' AND '1998–12-01')
Q5	SELECT * FROM STATEMENT WHERE Pid in (SELECT ID FROM PROPERTY WHERE PROPERTY = 'deathPlace' AND ns_id = (SELECT ID FROM NAMESPACE WHERE PRE-FIX = 'http://dbpedia.org/ontology') AND TS > '1900–01-01') LIMIT 500

It is also shown in Fig. 6 that, for storage of Dataset1, the vertical mapping and the PostSQL mapping take 0.69 min and 3.43 min, respectively. Their difference in storage time is not very significant because of the small size of the dataset. For storing a given dataset, the PostSQL mapping takes more time than the vertical mapping for two reasons. First, the PostSQL mapping needs to analyze the tRDF triples first to obtain the prefix and temporal information and then store the obtained information in different tables. tRDF data analysis inevitably leads to more time consumption. Second, the vertical mapping directly stores triples in one table and only executes the query once to judge the uniqueness of data. On the contrary, the PostSQL mapping stores triples in multiple tables and needs to check the uniqueness of each table, which must lead to the growth of time.

7.2.2 Query efficiency

Based on the PostSQL mapping method proposed in the paper, we used five query statements with different complexities to test their query efficiencies over datasets with different sizes. These five query statements are presented in Table 14.

In Table 14, Q1 is used to search all records in the Statement table of the PostgreSQL database and displays the first 500 records. Q2 is used to search all records with the subject "http://dbpedia.org/resource/Fiatau_Penitala_Teo." Q3 specifies the predicate on the basis of Q2, which is used to search the records with the predicate "http://xmlns.com/foaf/0.1/gender" and the subject "http://dbpedia.org/resource/Fiatau_Penitala_Teo." Q4 is used to search the records whose temporal information on the object starts on July 23, 1911, and ends between November 1, 1920, and December 1, 1998. Q5 is used to search the records whose predicate is "http://dbpedia.org/ontology/deathPlace," in which the temporal information on the predicate starts later than January 1, 1990, and displays the first 500 records. The experimental results of these five query statements are shown in Fig. 7.

Figure 7a, b, c, d, and e correspond to the query results of Q1 to Q5, respectively. First, it is shown in Fig. 7 that, for a query that does not involve temporal information, its query efficiency based on the two mapping methods is almost independent of the size of the dataset. At this point, with the vertical mapping, the maximum time differences of executing each of Q1, Q2, and Q3 on two datasets, Dataset4 and Dataset1, are respectively 39 ms, 23 ms, and 54 ms; with the PostSQL mapping, the maximum time differences of executing each of Q1, Q2 and

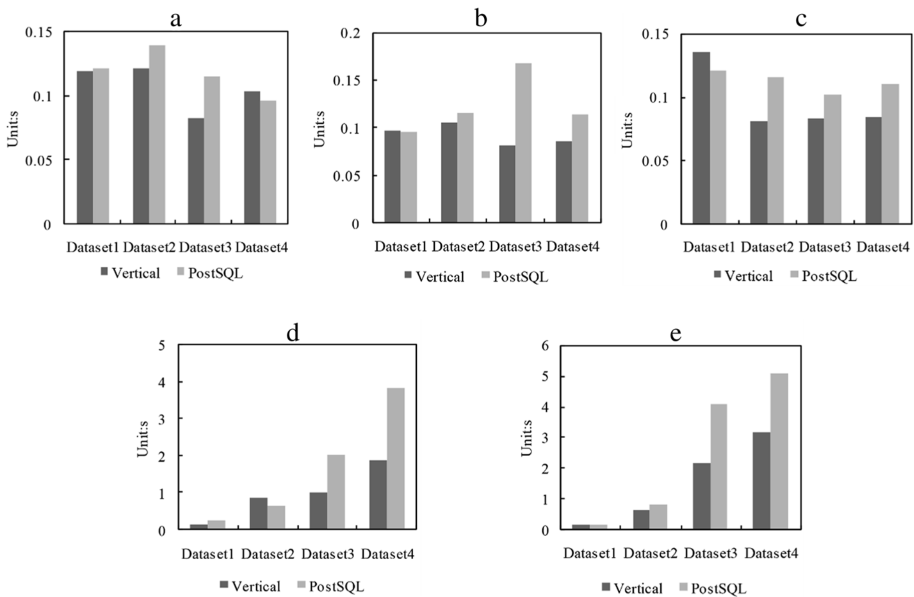

Fig. 7 Query time of different queries over datasets

Q3 on two datasets Dataset4 and Dataset1 are respectively 43 ms, 72 ms and 19 ms. For a query that involves temporal information, however, its execution time based on two mapping methods is significantly affected by the size of the dataset. For two queries Q4 and Q5, based on the PostSQL mapping, their execution time on Dataset4 are 16.21 times and 36.54 times longer than their execution time on Dataset1. In addition, by comparing Fig. 7d and e with Fig. 7a, b, and c, it can be observed that, for the same dataset, the queries involving temporal information generally need more execution time than the queries without temporal information. For the queries without temporal information, their executions based on the vertical mapping and the PostSQL mapping have an approximate query efficiency. For the queries with temporal information (e.g., Q4 and Q5), their executions based on the PostSQL mapping take more time. This is because PostSQL mapping uses multiple relational tables to store data and inevitably involves querying multi-tables. It is shown in Fig. 7d and e that, for queries Q4 and Q5 over Dataset4 with 10 million data, their executions based on the PostSQL mapping only take 1.95 s, 1.90 s longer than their executions based on the vertical mapping.

8 Conclusion and Future Work

In this paper, we proposed a novel temporal RDF model, termed tRDF, in which a temporal triple contains a temporal predicate or a temporal object. We defined the syntax and semantics of tRDF in detail. With the proposed temporal RDF model, we proposed the storage and query method to manage tRDF data with SQL:2011. We developed the rules and algorithms that can map tRDF data to relational databases. In addition, we extended the SPARQL query language and provided a temporal query language termed tSPARQLt for the tRDF model query. To efficiently query the temporal RDF data stored in the relational databases, we further defined partial transformation rules for transforming the query statements from tSPARQLt to SQL. Finally,

we designed comparison experiments to verify the feasibility and effectiveness of our proposed storage and query methods in terms of storage time and query efficiency.

In this paper, we used PostgreSQL to verify our storage and query methods against a temporal RDF dataset generated from DBPedia. In our future work, we plan to generate several massive temporal RDF datasets from, for example, Wikipedia and YAGO, and then verify our methods against these temporal RDF datasets with other relational database management systems such as Oracle, MySQL, and SQL Server. Considering that the queries with aggregations are usually used by professionals, we will investigate how to further extend tSPARQLt for aggregate queries over temporal RDF data and how to transform it to SQL. Scalability is a crucial issue with large-scale temporal RDF data management. In this direction, we will introduce optimization structures such as indexes and data partitioning and also explore the storage of massive temporal RDF data with NoSQL databases.

Acknowledgements The authors wish to thank the anonymous referees for their valuable comments and suggestions.

Authors' contributions Ruizhe Ma and Xiao Han completed the main manuscript text, Li Yan and Nasrullah Khan validated the experiments, and Zongmin Ma reviewed and edited the manuscript. All authors reviewed the manuscript.

Data Availability The data and materials used in the current study are available from the corresponding author on reasonable request.

Declarations

Ethical Approval Not applicable.

Competing interests All authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abadi, D. J., et al. (2009). SW-Store: A vertically partitioned DBMS for Semantic Web data management. *VLDB Journal,18*(2), 385–406. https://doi.org/10.1007/s00778-008-0125-y

Atre, M., & Hendler, J. A. (2009). *BitMat: a main memory bit-matrix of RDF triples. Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems 33–48*. CEUR-WS.org.

Bornea, M A *et al*. (2013), Building an efficient RDF store over a relational database, *Proceedings of the 2013 ACM International Conference on Management of Data*, ACM, 121–132. https://doi.org/10.1145/2463676.2463718

Brahmia, Z., et al. (2022). Ontology versioning driven by instance evolution in the τOWL framework. *Journal of Information & Knowledge Management,21*(1), 2250002:1-2250002:46. https://doi.org/10.1142/S0219649222500022

Brandt S *et al*. (2017), A framework for temporal ontology-based data access: a proposal. *Proceedings of the European Conference on Advances in Databases and Information Systems*, Springer, 161–173. https://doi.org/10.1007/978-3-319-67162-8_17

Canito, A., Corchado, J. M., & Marreiros, G. (2022). A systematic review on time-constrained ontology evolution in predictive maintenance. *Artificial Intelligence Review,55*(4), 3183–3211. https://doi.org/10.1007/s10462-021-10079-z

Chebotko, A., Lu, S., & Fotouhi, F. (2009). Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering,68*(10), 973–1000. https://doi.org/10.1016/j.datak.2009.04.001

Chen, L., et al. (2022). DACHA: A dual graph convolution based temporal knowledge graph representation learning method using historical relation. *ACM Transactions on Knowledge Discovery from Data,16*(3), 46:1-4618. https://doi.org/10.1145/3477051

Choi, P., Jung, J., & Lee, K. H. (2013). *RDFChain: chain centric storage for scalable join processing of RDF graphs using MapReduce and HBase. Proceedings of the ISWC 2013 Posters & Demonstrations Track 249–252*. CEUR-WS.org.

Clifford J and Croker A (1987), The historical relational data model (HRDM) and algebra based on lifespans. *Proceedings of the Third International Conference on Data Engineering*, IEEE, 528–537. https://doi.org/10.1109/ICDE.1987.7272420

Cudre-Mauroux P *et al*. (2013), NoSQL databases for RDF: an empirical evaluation. *Proceedings of the 12th International Semantic Web Conference*, Springer, 310–325. https://doi.org/10.1007/978-3-642-41338-4_20

Kalayci E G *et al*. (2018), Ontop-temporal: a tool for ontology-based query answering over temporal data. *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, ACM, 1927–1930. https://doi.org/10.1145/3269206.3269230

Erling O and Mikhailov I (2009), Virtuoso: RDF support in a native RDBMS, *Semantic Web Information Management* (eds. De Virgilio, R. *et al*.), Springer-Verlag, 501–519. https://doi.org/10.1007/978-3-642-04329-1_21

Faisal, S., & Sarwar, M. (2014). Temporal and multi-versioned XML documents: A survey. *Information Processing and Management.,50*(1), 113–131. https://doi.org/10.1016/j.ipm.2013.08.003

Fox A *et al*. (2013). Spatio-temporal indexing in non-relational distributed databases. *Proceedings of the IEEE International Conference on Big Data*, IEEE, 291–299. https://doi.org/10.1109/BigData.2013.6691586

Gadia, S. K. (1988). A homogeneous relational model and query languages for temporal databases. *ACM Transactions on Database Systems,13*(4), 418–448. https://doi.org/10.1145/49346.50065

Gao Q, *et al*. (2018), A semantic framework for designing temporal SQL databases. *Proceedings of the 37th International Conference on Conceptual Modeling*, Springer, 382–396. https://doi.org/10.1007/978-3-030-00847-5_27

Grandi, F. (2009). *Multi-temporal RDF ontology versioning, Proceedings of the 3rd International Workshop on Ontology Dynamics*. CEUR-WS.org.

Grandi, F. (2010). *T-SPARQL: a TSQL2-like temporal query language for RDF, Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems 21–30*. CEUR-WS.org.

Gutierrez, C., Hurtado, C. A., & Vaisman, A. A. (2007). Introducing time into RDF. *IEEE Transactions on Knowledge and Data Engineering,19*(2), 207–218. https://doi.org/10.1109/TKDE.2007.34

Hoffert, J., et al. (2013). YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence,194*, 28–61. https://doi.org/10.1016/j.artint.2012.06.001

Hogan, A., et al. (2010). *RDF needs annotations, Proceedings of 2010 W3C Workshop—RDF Next Steps*. W3C.

Hogan, A., et al. (2022). Knowledge graphs. *ACM Computing Surveys,54*(4), 1–37. https://doi.org/10.1145/3447772

Hu, Y., & Dessloch, S. (2015). Temporal data management and processing with column oriented NoSQL databases. *Journal of Database Management,26*(3), 41–70. https://doi.org/10.4018/JDM.2015070103

Huang, J. J., et al. (2020). Cluster query: A new query pattern on temporal knowledge graph. *World Wide Web,23*(2), 755–779. https://doi.org/10.1007/s11280-019-00754-1

Khadilkar, V., et al. (2012). Jena-HBase: a distributed, scalable and efficient RDF triple store. Proceedings of the 11th International Semantic Web Conference (Posters & Demos)85–88. CEUR-WS.org.

Koubarakis M and Kyzirakos K (2010), Modeling and querying metadata in the Semantic sensor Web: the model stRDF and the query language stSPARQL. *Proceedings of the 7th Extended Semantic Web Conference*, Springer, 425–439. https://doi.org/10.1007/978-3-642-13486-9_29

Kulkarni, K. G., & Michels, J.-E. (2012). Temporal features in SQL:2011. *ACM SIGMOD Record,41*(3), 34–43. https://doi.org/10.1145/2380776.2380786

Lee, K., & Liu, L. (2013). Scaling queries over big RDF graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment,6*(14), 1894–1905. https://doi.org/10.14778/2556549.2556571

Lopes N *et al*. (2010), AnQL: SPARQLing up annotated RDFS, *Proceedings of the 2010 International Semantic Web Conference*, Springer, 518–533. https://doi.org/10.1007/978-3-642-17746-0_33

Lu, W., Zhao, Z., & Wang, X. (2019). A lightweight and efficient temporal database management system in TDSQL. *Proceedings of the VLDB Endowment,12*(12), 2035–2046. https://doi.org/10.14778/3352063.3352122

Ma, Z. M., Capretz, M., & Yan, L. (2016). Storing massive Resource Description Framework (RDF) data: A survey. *The Knowledge Engineering Review,31*(4), 391–413. https://doi.org/10.1017/S0269888916000217

McBride, B. (2002). Jena: A Semantic Web toolkit. *IEEE Internet Computing,6*(6), 55–59. https://doi.org/10.1109/MIC.2002.1067737

Mckenzie, L. E., & Snodgrass, R. T. (1991). Evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys,23*(4), 501–543. https://doi.org/10.1145/125137.125166

Neumann, T., & Weikum, G. (2008). RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment,1*(1), 647–659. https://doi.org/10.14778/1453856.1453927

O'Connor, M. J., & Das, A. K. (2010). A lightweight model for representing and reasoning with temporal information in biomedical ontologies. Proceedings of the 3rd International Conference on Health Informatics 90–97. INSTICC Press.

Papailiou N *et al*. (2013), H2RDF+: High-performance distributed joins over large-scale RDF graphs, *Proceedings of the 2013 IEEE International Conference on Big Data*, IEEE, 255–263. https://doi.org/10.1109/BigData.2013.6691582

Perry M, Jain P and Sheth A P (2011), SPARQL-ST: Extending SPARQL to support spatiotemporal queries. *Geospatial Semantics and the Semantic Web*, Springer, 61–86. https://doi.org/10.1007/978-1-4419-9446-2_3

Pugiles A, Udrea O and Subrehmanian V S (2008), Scaling RDF with time. *Proceedings of the 17th International Conference on World Wide Web*, ACM, 605–614. https://doi.org/10.1145/1367497.1367579

Salas P E *et al* (2011), RDB2RDF plugin: relational databases to RDF plugin for eclipse. *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, ACM, 28–31. https://doi.org/10.1145/1984708.1984717

Shao B, Wang H X and Li Y T (2013), Trinity: a distributed graph engine on a memory cloud. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ACM, 505–516. https://doi.org/10.1145/2463676.2467799

Sintek M and Kiesel M (2006), RDFBroker: a signature-based high-performance RDF store. *Proceedings of the 3rd European Semantic Web Conference*, Springer, 363–377. https://doi.org/10.1007/11762256_28

Snodgrass, R. (1987). The temporal query language TQuel. *ACM Transactions on Database Systems,12*(2), 247–298. https://doi.org/10.1145/22952.22956

Snodgrass, R. (1994). TSQL2 language specification. *ACM SIGMOD Record,23*(1), 65–86. https://doi.org/10.1145/181550.181562

Stonebraker M R *et al*. (2005), C-Store: a column-oriented DBMS. *Proceedings of the 31st International Conference on Very Large Data Bases*, ACM, 553–564. https://doi.org/10.1145/3226595.3226638

Straccia U *et al.* (2010), A general framework for representing and reasoning with annotated Semantic Web data. *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. https://doi.org/10.1609/aaai.v24i1.7499

Tappolet J and Bernstein A (2009), Applied temporal RDF: efficient temporal querying of RDF data with SPARQL. *Proceedings of the 6th European Semantic Web Conference*, Springer, 308–322. https://doi.org/10.1007/978-3-642-02121-3_25

Udrea, O., Recupero, D. R., & Subrahmanian, V. S. (2010). Annotated RDF. *ACM Transactions on Computational Logic,11*(2), 1–41. https://doi.org/10.1145/1656242.1656245

Wang, H.-T., & Tansel, A. U. (2019). Temporal extensions to RDF. *Journal of Web Engineering,18*(1–3), 125–168. https://doi.org/10.13052/jwe1540-9589.18134

Wang Y F *et al* (2010), Timely YAGO: harvesting, querying, and visualizing temporal knowledge from wikipedia. *Proceedings of the 13th International Conference on Extending Database Technology*, ACM, 697–700. https://doi.org/10.1145/1739041.1739130

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment,1*(1), 1008–1019. https://doi.org/10.14778/1453856.1453965

Wolff, B. G. J., Fletcher, G. H. L., & Lu, J. J. (2015). An extensible framework for query optimization on TripleT-based RDF stores, Proceedings of the 2015 EDBT/ICDT Workshops 190–196. CEUR-WS.org.

Wu, G., et al. (2009). System II: A native RDF repository based on the hypergraph representation for RDF data model. Journal of Computer Science and Technology, 24 (4), 652–664. https://doi.org/10.1007/s11390-009-9265-9

Xuan D N, Bellatreche L and Pierra G (2006), A versioning management model for ontology-based data warehouses. *Proceedings of the 8th International Conference on Data Warehousing and Knowledge Discovery*, Springer, 195–206. https://doi.org/10.1007/11823728_19

Yan, L., Zhao, P., & Ma, Z. M. (2019). Indexing temporal RDF graph. *Computing,101*(10), 1457–1488. https://doi.org/10.1007/s00607-019-00703-w

Zaniolo, C., et al. (2018). User-friendly temporal queries on historical knowledge bases. *Information and Computation,259*(3), 444–459. https://doi.org/10.1016/j.ic.2017.08.012

Zhong Y, Fang J and Zhao X (2013), VegaIndexer: a distributed composite index scheme for big spatio-temporal sensor data on cloud. *Proceedings of the 2013 IEEE International Geoscience and Remote Sensing Symposium*, IEEE, 1713–1716. https://doi.org/10.1109/IGARSS.2013.6723126

Zhu, C C *et al*. (2021), Learning from history: modeling temporal knowledge graphs with sequential copy-generation networks. *Proceedings of the 2021 AAAI Conference on Artificial Intelligence*, AAAI, 4732–4740. https://doi.org/10.1609/aaai.v35i5.16604

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.