

Article

On the Implementations of the BiTemporal RDF Model: An Experimental Approach

Di Wu ^{1,*}, Hsien-Tseng Wang ¹ and Abdullah Uz Tansel ²¹ Lehman College, City University of New York, New York, NY 10468, USA; hsientseng.wang@lehman.cuny.edu² Baruch College, City University of New York, New York, NY 10010, USA; abdullah.tansel@baruch.cuny.edu

* Correspondence: di.wu@lehman.cuny.edu

Abstract

The BiTemporal RDF (BiTRDF) model extends the standard RDF data model by integrating both valid time and transaction time, thus enabling the representation and querying of dynamic and historical knowledge. While the theoretical foundations of BiTRDF have been established, practical implementation strategies have not yet been systematically studied. This paper bridges this gap by exploring six alternative approaches to implementing BiTRDF, combining object-oriented programming and database-oriented designs using Python and PostgreSQL. We evaluate these approaches using six synthetic datasets ranging from 0.5 million to 16 million bitemporal triples. The evaluation focuses on memory consumption, data-loading time, and query performance as data load increases. The results show that all approaches perform comparably when the knowledge store fits in memory. As the dataset size grows beyond available RAM, database-oriented implementations achieve substantially better loading and query performance, while object-oriented implementations offer greater flexibility and extensibility. These findings demonstrate the feasibility of implementing BiTRDF using existing technologies and provide practical guidance for selecting appropriate implementation strategies based on data size, performance requirements, and extensibility needs.

Keywords: Semantic Web; RDF; valid time; transaction time; bitemporal RDF

1. Introduction

The Internet has evolved beyond a platform for communication, transactions, and cloud storage. It is now a large and continuously expanding knowledge repository, where both humans and machines create, share, and analyze information. The Semantic Web was proposed to enable machine-understandable data and automated reasoning through the Semantic Web Layer Cake framework [1]. At the core of this framework lies the Resource Description Framework (RDF), which provides a standard model for representing knowledge as triples of the form $\langle \text{subject}, \text{predicate}, \text{object} \rangle$.

Despite its simplicity and widespread adoption, standard RDF is limited to representing binary relationships. Modeling more complex n-ary relationships often requires reification or additional extensions, which increase modeling complexity and may introduce performance overhead. These limitations become more pronounced when RDF is used to represent dynamic and evolving knowledge.

Temporal information is a fundamental aspect of many real-world applications, including legal records, financial systems, scientific data management, and historical knowledge



Academic Editor: Antony Bryant

Received: 25 December 2025

Revised: 12 April 2026

Accepted: 13 April 2026

Published: 15 April 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

bases. Temporal data can generally be classified into two types: *valid time*, which indicates when a fact holds in the real world, and *transaction time*, which records when that fact is stored, updated, or deleted in a data store. Temporal relational databases emphasize the importance of managing both dimensions of time to support auditing, historical queries, and data consistency.

Similarly, there is a growing need to support temporal semantics in RDF and RDF Schema. Temporal RDF extensions aim to capture the evolution of knowledge over time and enable historical reasoning. However, many existing approaches treat time as an attribute, rely heavily on reification, or support only a single temporal dimension. As a result, these models often suffer from increased structural complexity, limited extensibility, or unclear implementation strategies.

The BiTemporal RDF model (BiTRDF) addresses these challenges by uniformly extending RDF with both valid time and transaction time. In this model, time is treated as a reference, embedding temporal semantics directly into RDF resources rather than appending timestamps as auxiliary attributes. This design avoids the complexity of reification-based approaches for managing temporal knowledge [2,3]. In BiTRDF, all resources and relationships are inherently bitemporal, leading to cleaner semantics and stronger temporal consistency. While prior work has established the theoretical foundation of BiTRDF, practical questions remain open. In particular, it is unclear how BiTRDF can be efficiently implemented, stored, queried, and scaled using existing programming languages and data management technologies.

The goal of this paper is to bridge this gap between theory and practice. We explore multiple implementation strategies for BiTRDF by combining object-oriented programming techniques with database-oriented designs. Rather than proposing a single implementation, we investigate and compare six alternative approaches, each representing a different trade-off among flexibility, performance, memory consumption, and scalability. Through systematic experimentation, we evaluate these approaches using datasets of varying sizes and analyze their behavior under common operations such as data loading and querying.

To achieve the above goals, the following three considerations guided our study:

- Mapping BiTRDF effectively to the relational and object-oriented structures while maintaining temporal coherence.
- The performance trade-offs between memory-centric and storage-centric designs.
- Scaling various implementations when dataset sizes exceed physical memory.

The remainder of this paper is organized as follows. Section 2 reviews related work on temporal RDF extensions and bitemporal data management. Section 3 summarizes the key definitions and principles of the BiTemporal RDF model. Section 4 presents the design and implementation of the six BiTRDF approaches, including a running example and query illustrations. Section 5 describes the experimental setup and evaluation methodology, followed by a comparative analysis of performance results and a discussion of the findings in Section 6. Finally, Section 7 concludes the paper with recommendations and directions for future work.

Additional supporting details are provided in the Appendices: Appendix A details the input–process–output (IPO) workflow; Appendices B–G illustrate the running example as implemented across each of the six approaches; Appendix H provides comprehensive bitemporal query examples; and Appendix I contains the detailed data tables from the performance experiments.

2. Related Work

Although temporal RDF has been extensively studied, most research focuses on representation rather than efficient implementation. A common organization is where temporal

information is attached in the RDF model: Graph-level embedding, triple-level embedding, and resource-level embedding [4]. Graph-level approaches often use named graphs to associate a shared temporal context with a set of triples [5]. Triple-level approaches attach time to individual statements, frequently through explicit or implicit reification, which can increase the number of triples and complicate query processing [6,7]. Resource-level approaches temporalize RDF resources (subjects, predicates, and/or objects), for example, by temporalizing predicates (e.g., Singleton Property) or introducing entity versions (e.g., 4D Fluents) [8,9].

Instead of proposing a new encoding, this study implements and evaluates BiTRDF, a resource-level bitemporal model that assigns both time dimensions uniformly to RDF resources [2,3]. We therefore position our related work around (i) RDF store engineering and indexing, (ii) executable support for temporal/versioned RDF over existing engines, and (iii) bitemporal database foundations that inform implementation design.

2.1. RDF Store Implementations and Indexing for SPARQL Workloads

RDF query performance is primarily determined by physical design, including storage layout, indexing, and join processing. Sesame exemplifies a storage-independent architecture that separates query evaluation from the storage backend, enabling alternative backend realizations under the same query layer [10]. Surveys of RDF stores and SPARQL engines consolidate these implementation choices and emphasize that physical design is often the dominant performance factor [11].

Triple-pattern queries in RDF benefit from having indexes on multiple permutations of subject, predicate, and object. Hexastore supports this with sextuple indexing to speed up different access patterns [12]. RDF-3X likewise shows that careful index design, together with efficient query processing, can scale SPARQL evaluation to large datasets [13]. This motivates treating index and access-path design as key variables in our BiTRDF implementation study.

2.2. Executable Temporal and Versioned RDF over Existing RDF Stores

A key practical question is whether existing RDF standards can handle temporal semantics. Named graphs provide a standard way to attach metadata, such as provenance and temporal context, to sets of triples [5]. Using this idea, research on RDF archives typically encodes versions in the data and translates versioned queries into standard SPARQL systems. Cuevas and Hogan compare several named-graph encodings for RDF archives and study how these choices affect query cost [14]. Ostrich proposed storage and indexing methods for random-access versioned querying and analyzing the trade-offs among data loading, storage overhead, and query performance [15]. Overall, these studies show that time-aware RDF support often depends less on the abstract model and more on how the data is stored and how queries are executed.

2.3. Bitemporal Database for Implementing BiTRDF

BiTRDF is designed in the spirit of bitemporal databases. It supports both valid time and transaction time to enable operators analogous to temporal *as-of* and *change-oriented* queries. TSQL2 consolidates core temporal database concepts and standardizes temporal operators that later systems often adopt [16]. When these ideas are brought into RDF, many existing approaches either treat time as an attribute or add extra structural layers (e.g., reification or mapping constructs). SPARQL and its extensions, such as C-SPARQL, provide mechanisms for querying RDF data with temporal constraints, typically by evaluating time conditions at query execution. However, these approaches generally treat temporal information as annotations or external attributes and do not enforce temporal consistency at the data model level. In contrast, the BiTRDF model embeds valid-time and

transaction-time semantics directly into RDF resources and relationships, ensuring that temporal coherence is maintained as an intrinsic property of the data.

While several alternative temporal RDF models have been proposed, no established bitemporal RDF prototypes currently exist for side-by-side performance benchmarking [4]. The implementations presented in this work, therefore, focus on realizing this resource-level bitemporal representation, providing a storage and indexing foundation that can support temporal querying in a more consistent manner. We compare and evaluate trade-offs in storage organization, indexing, and query execution, based on established RDF store and archiving insights [11–13,15].

3. BiTRDF Model Overview

This section presents a concise overview of the *BiTemporal RDF (BiTRDF)* model [2,3], which provides the theoretical foundation for the implementation strategies examined later in this paper. This section introduces the key concepts and constraints that directly influence system design and implementation.

BiTRDF extends the standard RDF data model by integrating two temporal dimensions into all RDF resources. The central design principle of BiTRDF is that time is treated as a *reference* rather than an *attribute*. Instead of annotating facts with timestamps, temporal semantics are embedded directly into the structure of RDF resources and relationships. This design leads to a cleaner model and enables consistent reasoning about both real-world validity and system history.

BiTRDF introduces two temporal dimensions:

- Valid Time (VT), which represents the period during which a fact is true in the real world;
- Transaction Time (TT), which represents the period during which a fact is stored, known, or maintained in the knowledge base.

These dimensions enable BiTRDF to capture the evolution of both real-world facts and system knowledge, in a way that is conceptually aligned with bitemporal relational databases.

3.1. Bitemporal Resources and Triples

In BiTRDF, all RDF components—subjects, predicates, and objects—are inherently *bitemporal*. Each resource carries both a valid-time interval and a transaction-time interval, representing its real-world existence and its recorded history within the system.

- Bitemporal Resource (R^{bit}): A standard RDF resource r is extended with a valid-time interval (vt) and a transaction-time interval (tt). Formally, a bitemporal resource is defined as:

$$R^{bit} = \{r \bullet vt \bullet tt \mid r \in R \wedge vt \in VT_I \wedge tt \in TT_I\}, \quad (1)$$

where R denotes the set of RDF resources, and VT_I and TT_I denote the sets of valid-time and transaction-time intervals. Intuitively, vt specifies when a resource exists or holds in the real world, while tt specifies when it is known or maintained by the system.

- Bitemporal Triple ($TRIPLE^{bit}$): A bitemporal triple preserves the standard RDF structure while enforcing temporal consistency across all its components:

$$\begin{aligned} TRIPLE^{bit} = \{ \langle s^{bit} \ p^{bit} \ o^{bit} \rangle \mid & s^{bit} \in (I^{bit} \cup B^{bit}) \wedge p^{bit} \in P^{bit} \wedge \\ & o^{bit} \in (I^{bit} \cup B^{bit} \cup L^{bit}) \wedge \\ & p^{bit}.vt \neq \emptyset \wedge p^{bit}.tt \neq \emptyset \wedge \\ & p^{bit}.tt \subseteq (s^{bit}.tt \cap o^{bit}.tt) \wedge p^{bit}.vt \subseteq (s^{bit}.vt \cap o^{bit}.vt) \} \end{aligned} \quad (2)$$

where I^{bit} , B^{bit} , L^{bit} , and P^{bit} denote the sets of bitemporal IRIs, blank nodes, literals, and predicates, respectively.

This definition imposes a key constraint: a relationship can exist only when its subject and object coexist within the same valid-time and transaction-time intervals. This constraint is essential for maintaining temporal coherence and has direct implications for how BiTRDF data structures are stored and validated in practice.

3.2. Temporal Coherence and Core Operators

A central strength of the BiTRDF model is its built-in support for temporal coherence. BiTRDF ensures that all facts in a knowledge base are temporally consistent through an integrity rule and two core operators. These operators allow the system to reconstruct past system states and historical views of the real world.

- **BiTemporal Triple Integrity:** For a bitemporal predicate p^{bit} to be valid, its subject s^{bit} and object o^{bit} must coexist throughout the temporal span of the predicate:

$$p^{bit}.tt \subseteq (s^{bit}.tt \cap o^{bit}.tt) \wedge p^{bit}.vt \subseteq (s^{bit}.vt \cap o^{bit}.vt) \quad (3)$$

This rule prevents temporal contradictions, such as asserting relationships between entities that did not exist at the same time, and ensures that the knowledge base always represents a logically possible world.

- **Rollback Operator (δ_{t_I}):** The rollback operator reconstructs the state of the knowledge base at a past transaction time t_I :

$$\delta_{t_I}(KB) = \{f \mid f.tt \subseteq t_I\} \quad (4)$$

This operator supports queries such as “What did the system know at time t_I ?” and provides a formal mechanism for restoring historical system states.

- **Time Slice Operator (TS):** The time slice operator retrieves facts that were valid in the real world during a given valid-time interval i_I , as recorded at transaction time t_I :

$$TS(KB, i_I, t_I) = \{f \mid f.vt = i_I \wedge f.tt \subseteq t_I\} \quad (5)$$

This operator enables reconstruction of historical world states and supports queries such as “What was true during this period, based on what was known at that time?”

The complete formal semantics and proofs of these operators are provided in [2,3]. Together, the BiTRDF model and its core operators establish a rigorous and implementation-oriented foundation for managing bitemporal knowledge in RDF-based systems. In the next section, we build on this foundation to present concrete design choices and implementation strategies for BiTRDF.

4. Implementation Design

An RDF store is responsible for persistently storing, managing, and querying RDF graphs. Since BiTRDF extends the standard RDF model with bitemporal semantics, its implementation must support the creation, storage, retrieval, and querying of *bitemporal resources* and *bitemporal triples*, while preserving temporal coherence and acceptable query performance. Building upon the formal definitions of R^{bit} and $TRIPLE^{bit}$ introduced in Section 3, we now translate these theoretical constraints, particularly the Triple Integrity Rule, into concrete data structures and logic.

This section presents two implementation strategies for BiTRDF. The first strategy adopts a relational database perspective using PostgreSQL and emphasizes persistence,

indexing, and scalability. The second strategy follows an object-oriented programming (OOP) design implemented in Python (Version:3.8.8) and emphasizes flexibility, modularity, and in-memory performance. These approaches demonstrate trade-offs among performance, memory usage, extensibility, and system complexity. Particular attention is given to indexing and search mechanisms, since query processing is a central requirement of temporal knowledge stores.

4.1. Relational Database Implementation Using PostgreSQL

Relational database systems provide a mature and robust foundation for managing structured and persistent data. While native RDF stores exist, many systems cannot efficiently support complex interval-based logic [2]. In particular, most RDF engines are designed around fixed triple-based schemas and typically support temporal information through reification, named graphs, or annotation patterns, which introduce additional structural overhead and complicate query processing. PostgreSQL was specifically chosen due to its support of custom composite data types and GiST indexing, which are better suited for bitemporal interval arithmetic than standard triple-store indexing [17]. These features enable bitemporal attributes to be modeled directly and evaluated efficiently using native database operations, rather than being encoded indirectly within RDF structures. Furthermore, this choice enables precise control over storage layout and indexing strategies, which is essential for systematically evaluating the performance trade-offs of different BiTRDF implementations.

The database schema follows established principles of temporal database design [18] and maps BiTRDF concepts directly to relational constructs. The design consists of one composite data type and three core tables.

Bitemporal Resource Data Type: PostgreSQL allows users to define custom composite data types using the `CREATE TYPE` statement. To represent bitemporal resources, we define a composite type named *BiTemporalResource*, which encapsulates both valid-time and transaction-time intervals:

```
CREATE TYPE BiTemporalResource AS (
    Name VARCHAR,
    VtBegin TIMESTAMP WITH TIME ZONE,
    VtEnd   TIMESTAMP WITH TIME ZONE,
    TtBegin TIMESTAMP WITH TIME ZONE,
    TtEnd   TIMESTAMP WITH TIME ZONE
);
```

The *Name* field represents a standard RDF resource identifier, while the remaining fields capture its valid-time and transaction-time intervals. This data type directly corresponds to the BiTRDF resource model and is reused across tables to ensure consistency and simplify temporal query processing.

Bitemporal Triple Table: Bitemporal triples are stored in a dedicated table named *BiTemporalTripleTable*:

```
CREATE TABLE BiTemporalTripleTable (
    Id BIGINT PRIMARY KEY,
    Subject VARCHAR,
    Predicate BiTemporalResource,
    Object VARCHAR
);
```

Each row represents a single bitemporal triple. The *Id* column uniquely identifies a triple and supports indexing and future extensions. Subjects and objects are stored as resource identifiers, while predicates are stored using the *BiTemporalResource* type to capture

their temporal semantics. This design preserves the temporal integrity of relationships while avoiding unnecessary duplication of entity-level temporal data.

Entity and Predicate Tables: To manage temporal metadata for entities and predicates independently, two additional tables are defined:

```
CREATE TABLE BiTemporalEntityTable (
    Entity BiTemporalResource,
    PRIMARY KEY (Entity.Name)
);

CREATE TABLE BiTemporalPredicateTable (
    Predicate BiTemporalResource,
    PRIMARY KEY (Predicate.Name)
);
```

These tables store the temporal lifespans of subjects, objects, and predicates. Primary key constraints enforce uniqueness and allow efficient lookup of bitemporal resources during query execution and integrity validation.

Indexing and Query Efficiency: Efficient query processing is critical for RDF stores. PostgreSQL automatically creates B-tree indexes for primary keys, enabling fast retrieval of entities and predicates. Additional indexes are defined on the triple table to support common RDF access patterns:

```
CREATE INDEX SP0index ON BiTemporalTripleTable
    (Subject, Predicate.Name, Object);

CREATE INDEX PS0index ON BiTemporalTripleTable
    (Predicate.Name, Subject, Object);

CREATE INDEX OPSindex ON BiTemporalTripleTable
    (Object, Predicate.Name, Subject);
```

These indexes correspond to standard subject–predicate–object permutations and allow queries constrained by any triple component to be evaluated efficiently. B-tree indexes provide logarithmic-time complexity and support both exact matching and range-based temporal conditions.

4.2. Object-Oriented Implementation Using Python

In addition to the database-oriented approach, BiTRDF can be implemented using an object-oriented design in Python. This approach emphasizes modularity, extensibility, and ease of experimentation, making it well-suited for prototyping and in-memory analytics.

Input Processing and Parsing: The implementation reads BiTRDF graphs expressed in Turtle syntax [19]. Each input line represents a bitemporal triple. The input processor parses the file sequentially and decomposes each line into RDF resources, temporal intervals, and structural relationships. For simplicity, input files are assumed to be syntactically correct.

Construction of Bitemporal Components: The object-oriented design decomposes BiTRDF construction into several modular components:

- Resource Constructor, which converts IRIs, literals, or blank nodes into RDF resource objects;
- Temporal Data Constructor, which extracts temporal annotations and constructs valid-time and transaction-time intervals;
- Bitemporal Resource Constructor, which combines RDF resources with temporal intervals;
- Bitemporal Triple Constructor, which assembles bitemporal subjects, predicates, and objects while enforcing BiTemporal Triple Integrity.

This modular structure makes it straightforward to extend the system with additional dimensions, such as confidence values or spatial attributes. The overall input–process–output workflow is summarized in Table A1.

Query Processing and Serialization: A unified query processor supports pattern-based searches over bitemporal triples. Queries may constrain subjects, predicates, objects, valid time, transaction time, or combinations of these dimensions. The processor is designed to be generic and parameter-driven, enabling flexible query formulation.

The system also supports serialization, allowing stored knowledge to be exported back into Turtle format. This ensures consistency between input and output representations and supports interoperability with other RDF tools. The BiTRDF model is defined at the conceptual level and is independent of any specific RDF serialization format. While Turtle is used in our implementation for readability, the underlying bitemporal semantics can be preserved in alternative formats such as N-Triples or JSON-LD, provided that valid-time and transaction-time intervals are explicitly encoded as part of the resource representation. In this implementation, temporal intervals are represented as structured literal values, which can be consistently serialized across formats. As long as the temporal components and integrity constraints of BiTRDF are maintained, the choice of serialization does not affect the correctness of the model.

In-Memory Indexing with Dictionaries: To support fast in-memory queries, the OOP implementation uses Python dictionaries as indexing structures. Dictionaries provide hash-based access with constant-time lookup on average. Three nested dictionary indexes are maintained:

- Subject-indexed: $\{s^{bit} : \{p^{bit} : \{o^{bit}\}\}\}$
- Predicate-indexed: $\{p^{bit} : \{o^{bit} : \{s^{bit}\}\}\}$
- Object-indexed: $\{o^{bit} : \{s^{bit} : \{p^{bit}\}\}\}$

This design enables efficient queries but increases memory usage and update overhead because all indexes must remain synchronized. This reflects a common space–time trade-off in knowledge store design.

4.3. Summary of Design Trade-Offs

The PostgreSQL-based implementation emphasizes persistence, scalability, and mature indexing support, making it suitable for large and long-lived BiTRDF knowledge stores. In contrast, the object-oriented implementation emphasizes flexibility, extensibility, and fast in-memory access, making it well-suited for experimentation and smaller-scale workloads. These complementary designs form the basis for the empirical evaluation presented in the next section.

5. Experiments

The goals of the experimental evaluation are threefold: (1) to demonstrate that BiTRDF knowledge stores can be correctly constructed and queried; (2) to validate the correctness of bitemporal querying under valid-time and transaction-time constraints; (3) to compare the performance of different implementation alternatives in terms of memory usage, loading time, and query efficiency.

5.1. Triple Stores Available

We examined a wide range of existing triple stores, including RDF Benchmarks [20], Oracle Spatial and Graph 12c [21,22], AllegroGraph [23], Stardog [24–26], OpenLink Virtuoso [27–29], GraphDB [30,31], BlazeGraph [32], Jena TDB [33–35], RDF4Led [36], RDFox [37], RDFLib [38], 3Store [39], 4Store [40], OpenRDF Sesame [41], Rya [42]. While these triple stores support the standard RDF model and often integrate SQL capabilities,

extending their resources to include temporal information remains difficult. While these existing solutions satisfy some of the expectations, they are missing the crucial need of extensibility to include temporal knowledge.

5.2. Implementing Alternatives

Based on the design premises described in Section 4, we developed a software package named *BiTRDFLib* that includes six implementation alternatives. These alternatives differ in how BiTRDF elements are represented and stored, allowing us to explore trade-offs between flexibility, performance, and scalability. The alternatives are grouped into two categories: object-oriented implementations and storage-oriented implementations.

The object-oriented (OOP) implementations explicitly construct BiTRDF elements, including temporal intervals, bitemporal resources, and bitemporal triples. All OOP alternatives share the same parsing, construction, querying, and serialization logic, and differ only in their underlying storage structures.

- OOP + Dictionary: Uses Python dictionaries with hash-based indexing. Three dictionaries are maintained to optimize subject-, predicate-, and object-based queries.
- OOP + DataFrame: Uses Pandas DataFrames to store bitemporal triples, entities, and predicates in tabular form, leveraging built-in indexing and filtering operations.
- OOP + List: Stores each bitemporal triple as a list element without reorganization. This approach minimizes structural overhead but does not optimize query performance.

The storage-oriented implementations prioritize simplicity and performance by directly storing atomic values rather than object instances. These approaches bypass the construction of BiTRDF elements and instead focus on efficient value-level storage.

- Database + PostgreSQL: Uses a relational schema with composite bitemporal data types, as described in Section 4, and persists data in a PostgreSQL database.
- Database + Dictionary: Stores each bitemporal triple as 15 atomic values corresponding to the three bitemporal resources and their temporal intervals, using a Python dictionary.
- Database + List: Stores each bitemporal triple as a fixed-size list of 15 elements, enabling direct index-based access with minimal overhead.

5.3. Running Example BiTRDF Knowledge Store

For demonstration purposes, we used a running example BiTRDF knowledge store:

```
:NJ[0,now][0,now] bitrdf:type[0,now][0,now] :state[0,now][0,now] .
:NY[0,now][0,now] bitrdf:type[0,now][0,now] :state[0,now][0,now] .
:CO[0,now][0,now] bitrdf:type[0,now][0,now] :state[0,now][0,now] .
:NYC[0,now][0,now] bitrdf:type[0,now][0,now] :city[0,now][0,now] .
:CUNY[0,now][0,now] bitrdf:type[0,now][0,now] :Univ[0,now][0,now] .
:CU[0,now][0,now] bitrdf:type[0,now][0,now] :Univ[0,now][0,now] .
:NYC[0,now][0,now] :locatesAt[0,now][0,now] :NY[0,now][0,now] .
:CUNY[0,now][0,now] :locatesAt[0,now][0,now] :NY[0,now][0,now] .
:CU[0,now][0,now] :locatesAt[0,now][0,now] :CO[0,now][0,now] .
:Lu[1,now][1,now] bitrdf:type[1,now][1,now] :person[0,now][0,now] .
:Ab[1,now][1,now] bitrdf:type[1,now][1,now] :person[0,now][0,now] .
:Lu[1,now][1,now] :livesIn[1,5][1,5] :NJ[0,now][0,now] .
:Lu[1,now][1,now] :studyAt[1,now][1,now] :CUNY[0,now][0,now] .
:Lu[1,now][1,now] :livesIn[5,10][5,10] :NY[0,now][0,now] .
:Ab[1,now][1,now] :livesIn[5,10][5,10] :NJ[0,now][0,now] .
:Lu[1,now][1,now] :livesIn[10,now][10,now] :CO[0,now][0,now] .
:Lu[1,now][1,now] :worksAt[10,now][10,now] :CU[0,now][0,now] .
:Ab[1,now][1,now] :livesIn[10,now][10,now] :NY[0,now][0,now] .
:Lu[1,now][1,now] :worksWith[10,now][10,now] :Ab[1,now][1,now] .
```

5.4. Object-Oriented Programming Implementation

The three OOP-based alternatives differ only in storage, sharing all other processing logic. We explain the different storage processes of each alternative separately in detail in this section.

OOP + Dictionary: In this alternative, we adopted the dictionary data structure in Python. The dictionary data structure uses a {key:value} pair to organize data. Using hashing, the access time for the value associated with a key is $O(1)$. To maximize the searching time for queries, three dictionaries are created to store the BiTRDF graph in different ways. Once a bitemporal triple is constructed, its subject, predicate, and object components will be saved as:

```
subject_dictionary = {subject : {predicate : {object}}}  
predicate_dictionary = {predicate : {object : {subject}}}  
object_dictionary = {object: {subject : {predicate}}}
```

The running example stored in dictionaries is depicted in Appendix B.

To speed up the response time for queries, we store the same bitemporal triple in three dictionaries that are optimized for the subject, the predicate, or the object of triples. If a query is about a particular component of bitemporal triples, the corresponding dictionary will be called for the query. If the AS OF clause is not specified, the default *now* will be used.

OOP + DataFrame: In this alternative, we adopted the DataFrame structure of the Pandas package. Pandas is a widely used package to deal with tabular data, and DataFrame is well known for its fast and efficient data manipulation with integrated indexing. When a bitemporal triple is constructed, it will be stored in three dataframes.

```
bitemporalTriple_df=DataFrame({Subject.r:[], Predicate:[], Object.r:[]})  
bitemporalEntity_df=DataFrame({Entity:[]})  
bitemporalPredicate_df=DataFrame({Predicate:[]})
```

The running example stored in dataframes is depicted in Appendix C.

OOP + List: In this alternative, we adopted the list data structure in Python. The list is very basic but powerful. Once a bitemporal triple is constructed, it will be saved directly as an element in a list without any modification. Thus, if the BiTRDFLib package needs extensions, modifying the underlying structure of triples will not affect the storing and query processes. This alternative is included to provide a straightforward implementation for scenarios where rapid deployment is prioritized over efficiency.

```
bitemporal_graph = [bitemporal_triples]
```

The running example stored in a list is depicted in Appendix D.

5.5. Database Implementation

While OOP alternatives construct BiTRDF instances, database-oriented ones focus on retrieval of triples. Without the BiTRDF elements, alternatives in the database logic lack flexibility and extensibility in the OOP logic, but they can perform more efficiently both in loading and searching. We discuss the possible alternatives in detail in this section.

Database + PostgreSQL: In this alternative, we adopted a relational database. We created a PostgreSQL database, and used Pony, a Python package, to communicate between Python and PostgreSQL. The running example stored in a relational table is depicted in Appendix E.

Database + Dictionary: In this alternative, we adopted Python dictionary data structure as well. A bitemporal triple has 15 parts, respectively:

- *subject_resource, subject_vt_s, subject_vt_e, subject_tt_s, subject_tt_e,*
- *predicate_resource, predicate_vt_s, predicate_vt_e, predicate_tt_s, predicate_tt_e,*
- *object_resource, object_vt_s, object_vt_e, object_tt_s, object_tt_e*

This storage structure is a direct mapping of the formal BiTemporal Resource definition. Since a single bitemporal resource consists of 5 components, and a triple consists of 3 bitemporal resources, the total storage requirement is 15 atomic values. Since each line in the text file represents a bitemporal triple, we retrieve and store these 15 values in a dictionary directly. The running example stored in a dictionary is depicted in Appendix F.

Database + List: In this alternative, we adopted the Python list data structure as well. As we discussed in the previous alternative, each bitemporal triple contains 15 parts. A list of size 15 can store the values of these parts. The running example stored in a list is depicted in Appendix G.

5.6. Query Examples

We evaluate three classes of queries:

- Structural queries, which filter by subject, predicate, or object;
- Valid-time queries, which retrieve facts valid at a specified time instant;
- Transaction-time queries, which reconstruct the knowledge store as of a given transaction time.

Representative query results are shown in Tables A2–A13. These examples confirm that BiTRDF operators correctly enforce valid-time constraints and support rollback semantics.

5.7. Use-Case Scenario

The dataset is designed to simulate dynamic knowledge graphs with temporal evolution, where entities and relationships change over both real-world time (valid time) and system recording time (transaction time). Such scenarios commonly arise in domains including:

- Personnel and employment history tracking,
- Location and mobility data (e.g., individuals moving across regions),
- Organizational affiliations and institutional changes,
- Versioned knowledge bases and data auditing systems.

In this study, we use this temporal entity–relationship evolution scenario as an abstracted use case for domains such as personnel records, mobility tracking, and organizational history, where entities change locations and affiliations over time and those changes must be audited with respect to both valid time (when changes occur in the real world) and transaction time (when they are recorded). This conceptual scenario directly informs the design of our synthetic datasets, which encode similar patterns of evolving entities and relationships under both temporal dimensions, while allowing us to systematically control scale and temporal distributions for performance experiments. The running example in Section 5.3 (e.g., individuals living in different states over time and working at institutions) reflects a temporal entity–relationship evolution scenario, which serves as a conceptual grounding for the larger synthetic datasets.

5.8. Dataset Description

To evaluate performance and scalability, we generated six synthetic BiTRDF knowledge stores. The datasets are synthetically generated to systematically control:

- The number of bitemporal triples;
- The distribution of valid-time and transaction-time intervals;
- The independence of structural and temporal dimensions.

This design allows us to isolate the system-level performance characteristics (memory usage, loading time, and query efficiency) without confounding factors introduced by domain-specific data irregularities. While the data is synthetic, it preserves the structural and temporal properties of real-world temporal knowledge graphs, including overlapping temporal intervals, evolving relationships, and multi-version entity states. Building on this controlled design, the datasets are further structured to evaluate scalability across increasing data volumes. Specifically, we generated six knowledge stores of progressively larger sizes to simulate high-frequency bitemporal event scenarios, such as financial auditing or legal logging, where precise tracking of both valid time and transaction time is critical for data provenance. Each bitemporal triple is composed of three such resources. For example, the following are randomly generated bitemporal triples:

```
...
:wz [45,96) [43,68) :ci [51,85) [43,68) :tp [51,85) [1,68) .
:mx [67,82) [4,85) :tk [68,72) [46,59) :qb [55,78) [34,84) .
:ey [0,85) [48,96) :dh [49,53) [48,61) :ip [49,90) [17,61) .
:hm [60,75) [21,96) :ln [60,65) [65,66) :hd [12,70) [14,84) .
:sr [7,18) [1,87) :ac [18,25) [6,22) :fe [8,44) [3,82) .
:ac [21,41) [3,44) :qj [21,39) [13,40) :hl [13,93) [3,40) .
:gy [23,37) [1,52) :aa [26,29) [15,26) :jw [9,65) [1,85) .
:dy [6,88) [2,50) :df [20,83) [2,32) :jf [16,92) [2,38) .
:sj [4,13) [1,18) :dq [5,10) [1,10) :ao [0,18) [1,12) .
:af [6,22) [12,65) :ai [9,13) [12,46) :yt [5,98) [11,86) .
:qt [12,42) [0,3) :sc [12,24) [0,3) :lv [6,100) [0,20) .
...
```

BiTRDF knowledge stores are created based the specified number of bitemporal triples. We started with a relatively small knowledge store, with only 500,000 triples, and ended with a relatively large knowledge store, with 16,000,000 triples. The number of triples was doubled for each triple store to evaluate the scaling of their performance. In particular, we would like to examine and compare the performance before and after the computation needs exceed the Random Access Memory (RAM) and the operating system has to swap between RAM and physical storage. We summarize the six BiTRDF knowledge stores in Table 1.

Table 1. Summary of the six knowledge stores.

Name	Number of Bitemporal Triples	Size
D1	500,000	37 MB
D2	1,000,000	73 MB
D3	2,000,000	146 MB
D4	4,000,000	292 MB
D5	8,000,000	583 MB
D6	16,000,000	1170 MB

The chosen dataset design is particularly suitable for evaluating the BiTRDF model. First, it requires the simultaneous management of valid time (real-world changes) and transaction time (system-recorded history), which are the two core temporal dimensions of BiTRDF. Second, the generated data enforces temporal consistency across entities and relationships, reflecting the integrity constraints defined in the BiTRDF model (Section 3). Third, the dataset supports representative query patterns, including time-slice queries (e.g., “what was true at time t?”) and rollback queries (e.g., “what did the system know at time t?”), which are directly aligned with the BiTRDF operators. Therefore, the dataset

provides a controlled yet representative environment for evaluating both the correctness and performance of BiTRDF implementations.

5.9. Testing Device Description

All experiments were conducted on a consumer-grade laptop with the configuration shown in Table 2. This setup reflects a realistic deployment environment rather than a specialized server.

Table 2. Summary of the Testing Device.

Tech Specification	Value
Brand	Apple
Model	MacBook Pro, 13-inch, 2020
CPU	M1
Memory	16 GB
OS	macOS Monterey
Python	3.8.8

5.10. Evaluation Criteria

We measured performance using three standardized criteria: Memory Consumption, Loading Time, and Query Time.

Memory Consumption: BiTRDFLib first loads data into memory to construct and store a BiTRDF knowledge store. Memory consumption reflects the space efficiency of processing and storing a BiTRDF knowledge store. Memory consumption is measured using *psutil*, a cross-platform library for retrieving information on running processes and system utilization. This measurement compares the total memory usage before and after the execution of a function. It can also indicate when the RAM is not big enough to handle the needs: when the swap between RAM and physical storage happens, we can observe a small or even a negative memory consumption.

Loading Time: When comparing different implementations, loading time is part of overall performance. Because all data must be loaded into memory before query tasks can begin, the time required for loading directly impacts the efficiency of the system.

Query Time: In practice, search time is critical to the performance of the implementation, since searches are executed frequently by many users. Long response time is unacceptable for a satisfactory user experience. It is important for us to understand how each alternative performs in this criterion. It is also important to compare performance before and after the swap has happened, since it is directly associated with scalability. We used a simple query to find all triples whose subjects' resource matches, for the evaluation.

The time used for loading and searching is measured using *cProfile*, a Python built-in module that provides statistics on the execution of a program. In order to obtain a more accurate result, we computed the mean of these quantitative values across 100 runs for each alternative. In addition to the total value of these three quantitative facts, we also calculated their per-million-triple values, as they are useful for evaluating scalability.

6. Results and Discussion

This section presents and analyzes the experimental results from three perspectives: memory consumption, loading time, and query time. The goal is to understand how different implementation alternatives behave as the size of the BiTRDF knowledge store increases, and to identify the trade-offs between extensibility and performance.

6.1. Memory Consumption

Table A14 summarizes the memory consumption of all six implementation alternatives across the six datasets. For smaller knowledge stores (D1 and D2), all alternatives exhibit comparable memory usage. At this scale, memory consumption is mainly influenced by implementation overhead rather than dataset size. The PostgreSQL-based alternative consumes more memory due to the initialization of a separate database process and internal buffering.

When the dataset reaches D3, PostgreSQL begins to offload data to disk, resulting in a temporary reduction in observed memory usage. This behavior reflects internal database memory management rather than improved space efficiency. For larger datasets (D4–D6), all alternatives exceed available RAM and rely on disk swapping to some extent.

Among the OOP-based alternatives, memory usage remains relatively stable after swapping occurs, although object creation and metadata introduce nontrivial overhead. In contrast, the database-oriented dictionary and list alternatives show higher memory growth at larger scales because all atomic values are stored explicitly without object reuse.

Figure 1 shows both absolute memory usage and per-million-triple trends. Overall, memory consumption is primarily driven by data representation choices and system-level memory management once the dataset size exceeds physical memory.

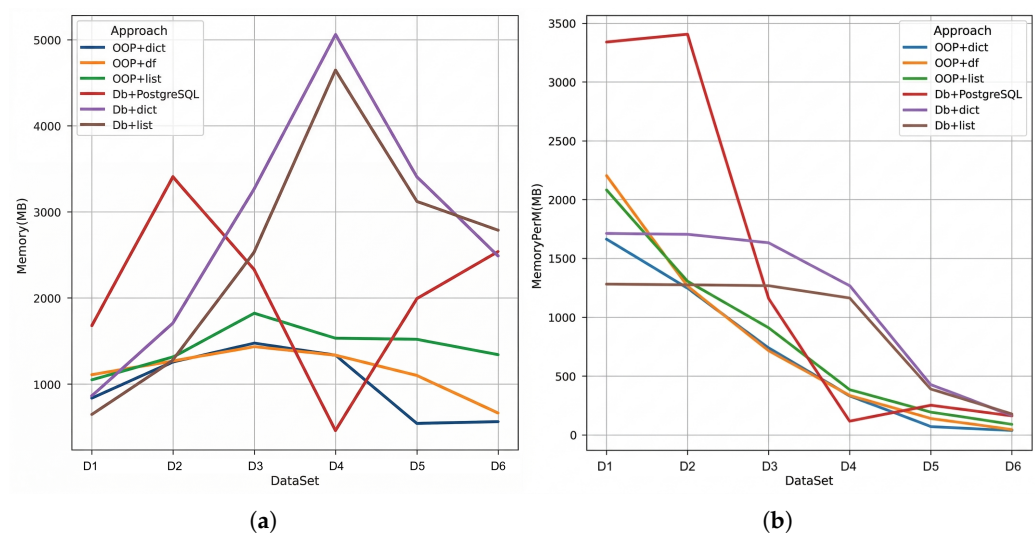


Figure 1. Memory consumption results. (a) Memory trend across datasets. (b) Per-million-triple memory trend.

6.2. Loading Time

Table A15 reports details of the time required to load each knowledge store. Loading time increases with dataset size for all alternatives, as expected. For small and medium datasets (D1–D3), the differences are moderate, and all implementations complete loading within a reasonable time. Once the dataset exceeds available memory (D4–D6), performance gaps become significant.

The OOP-based alternatives incur higher loading costs because each triple must be parsed, instantiated as objects, and connected through internal structures. This overhead grows rapidly under memory pressure. Among these alternatives, the list-based implementation performs best due to its simpler storage model.

Database-oriented alternatives load data more efficiently by storing atomic values directly. The dictionary-based database approach consistently achieves the fastest loading times, followed by the list-based variant. PostgreSQL shows stable behavior but introduces overhead from database communication and transaction handling.

These trends are clearly illustrated in Figure 2, especially in the per-million-triple comparison. Overall, database-oriented storage is better suited for large-scale data ingestion, while OOP-based implementations favor semantic clarity over loading efficiency.

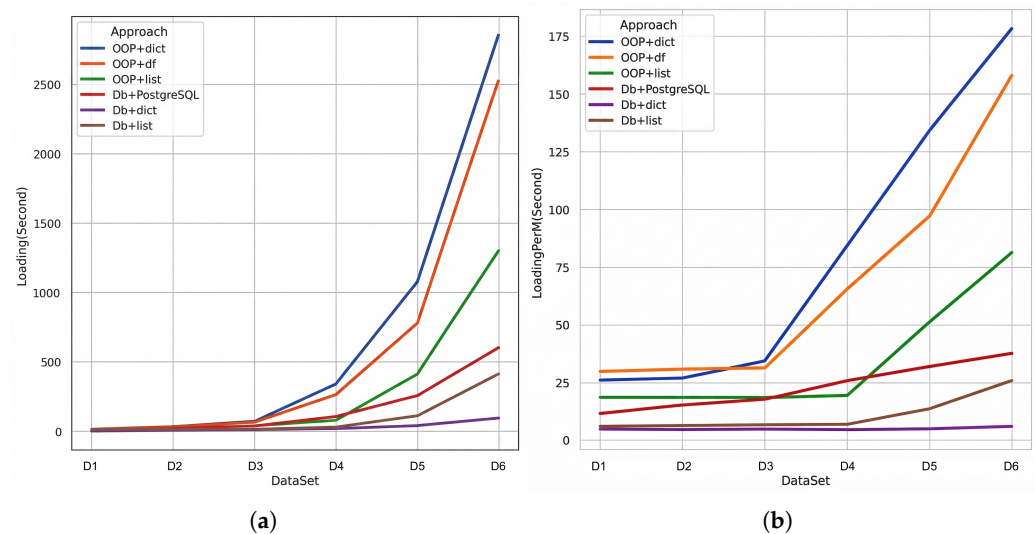


Figure 2. Loading time results. (a) Loading time trend across datasets. (b) Per-million-triple loading trend.

6.3. Query Time

Table A16 summarizes the details of the query time results. For small datasets (D1 and D2), all alternatives provide fast query response times. As the dataset size increases, performance differences become more apparent. The OOP + dictionary alternative shows a sharp increase in query time from D3 onward, mainly due to nested dictionary traversal and memory swapping effects.

Other alternatives scale more smoothly. The list-based OOP approach maintains acceptable performance despite linear scans, benefiting from minimal structural overhead. Database-oriented alternatives perform consistently, with PostgreSQL benefiting from internal query optimization and efficient disk access.

Per-million-triple results indicate that query time grows approximately linearly for most alternatives, suggesting stable scalability. Figure 3 confirms that database-oriented implementations offer more predictable query performance under memory pressure.

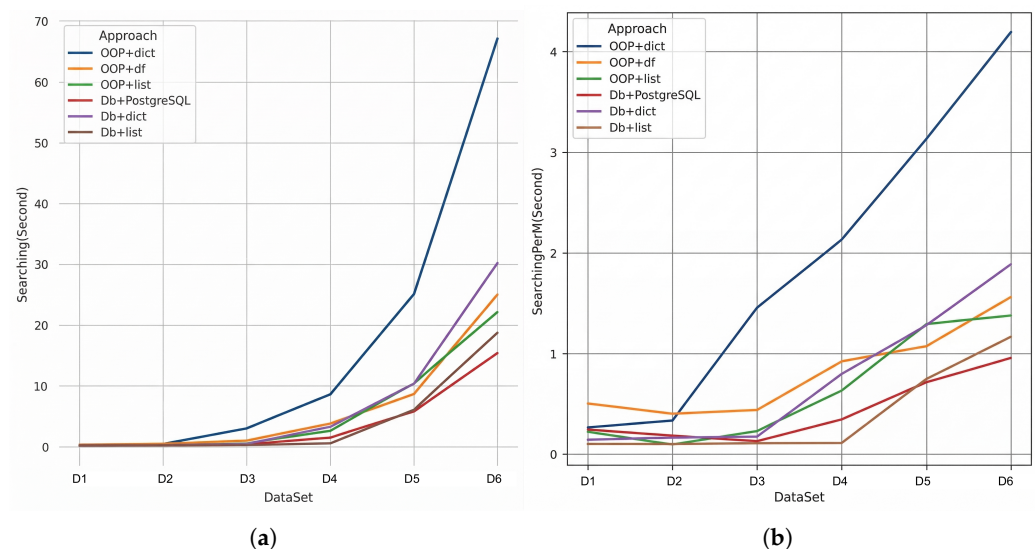


Figure 3. Query time results. (a) Query time trend across datasets. (b) Per-million-triple query trend.

Overall, the results reveal a clear trade-off between extensibility and performance. OOP-based implementations closely follow the BiTRDF conceptual model and are well-suited for experimentation and future extensions. Database-oriented implementations, in contrast, provide better scalability and efficiency for large-scale loading and querying tasks.

7. Conclusions

This paper presents the first systematic study of practical implementation methods for the BiTemporal RDF model. Through a comprehensive experimental evaluation of six implementation alternatives, we examined how different design choices affect memory consumption, loading time, and query performance across knowledge stores of increasing size.

The results show that, for small-scale knowledge stores, all alternatives perform similarly. When sufficient RAM is available (up to approximately one million triples), loading times remain under 30 s and query times remain below 0.5 s for all implementations. At this scale, the choice of implementation has little impact on overall system performance, allowing developers to prioritize design clarity and extensibility.

Performance differences become pronounced once the knowledge store exceeds available memory. For datasets larger than two million triples, database-oriented implementations consistently outperform object-oriented ones in both loading and querying. At the largest scale tested (16 million triples), the OOP + Dictionary alternative requires 2849 s to load the data, while the Database + Dictionary alternative completes loading in only 91.9 s. A similar gap is observed in query performance, where OOP + Dictionary requires 67 s, compared to 15.25 s for the PostgreSQL-based implementation. The results show that the storage strategy is critical for scalability.

Beyond quantitative performance, the architectural trade-offs among the alternatives are critical. While database-oriented designs favor retrieval efficiency, the Object-oriented implementations closely align with the BiTRDF conceptual model and provide strong flexibility and extensibility. These methods allow developers to modify bitemporal resource definitions, introduce additional dimensions, or adapt parsing and serialization logic with minimal effort. For applications that operate on moderate-sized knowledge stores or emphasize rapid development and experimentation, these methods offer a practical and maintainable solution.

Unlike SPARQL and its extensions, such as C-SPARQL, which primarily address temporal aspects at the query level through filtering or stream processing, our implementation focuses on embedding bitemporal semantics directly at the data model and storage level. This resource-level approach avoids reliance on reification or auxiliary structures to represent temporal information, thereby reducing structural overhead and improving efficiency. As a result, the proposed implementation provides a practical storage and indexing foundation that can support existing or future temporal SPARQL extensions.

The main contributions of this paper are summarized as follows:

- Presenting the first comprehensive study of practical implementation strategies for the BiTemporal RDF model.
- Designing and implementing six alternative approaches for storing and managing BiTRDF knowledge stores using Python and PostgreSQL.
- Empirically evaluating these approaches based on memory consumption, data loading time, and query performance across multiple dataset scales.
- Analyzing the strengths and limitations of each approach and providing practical guidance for selecting suitable implementation strategies under different system requirements.

Rather than delivering a fully production-ready BiTRDF system, this study demonstrates the feasibility of the BiTRDF model and highlights key design considerations for

its implementation. Several directions remain open for future work, including support for update and deletion operations, conversion between BiTRDF and standard RDF, integration with SPARQL-based querying, and the development of temporal inference mechanisms. Together, these extensions would further advance the adoption of bitemporal semantics in RDF-based knowledge systems.

Author Contributions: Conceptualization of implementation alternatives and writing—original draft preparation, D.W.; writing—review and editing, D.W. and H.-T.W.; development of the BiTRDF model theory and supervision, A.U.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The generated bitemporal knowledge stores are shared via URL https://osf.io/q3yvw/overview?view_only=d9502a05353f4ff19dbb95e492ae86e7 (accessed on 22 December 2025) .

Acknowledgments: During the preparation of this manuscript/study, the authors used ChatGPT 5 and Gemini 3 for the purposes of proofreading. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RDF	Resource Description Framework
BiTRDF	BiTemporal RDF
OOP	Object-Oriented Programming
SQL	Structured Query Language
SPARQL	SPARQL Protocol and RDF Query Language
IPO	Input-Process-Output
RAM	Random Access Memory

Appendix A. Input-Process-Output

Table A1. IPO of Processors, Constructors, Output, and Serialization.

Function	Input	Process	Output
Input	The path to a text file which contains a BiTRDF graph written in Turtle syntax [19]	Process a BiTRDF graph written in Turtle syntax: (1) read the text file line by line; (2) call the resource constructor, temporal data constructor, bitemporal resource constructor, bitemporal triple constructor to break down each line to basic elements and to constructor a bitemporal triple from each line	A BiTRDF graph
Resource	A string that represents an IRI, a literal, or a blank node	Construct a standard RDF resource	A standard RDF resource

Table A1. Cont.

Function	Input	Process	Output
Temporal data	A string that represent a time interval which starts with "[", has a lower bound t_s and an upper bound t_e , and ends with "]" if t_e is <i>now</i> or ")" if t_e is not <i>now</i>	Construct a time interval	A time interval as $[t_s, now]$ or $[t_s, t_e]$
BiTemporal Resource	A standard RDF resource, r , a valid time interval, and a transaction time interval	Concatenate the resource component, valid time component, and transaction time component together to create a bitemporal resource	A bitemporal resource r^{bit}
BiTemporal Triple	Three bitemporal resources, s^{bit} , p^{bit} , and o^{bit}	Place the three resources in the corresponding positions of a triple as the subject, the predicate, and the object	A bitemporal triple, $\langle s^{bit} p^{bit} o^{bit} \rangle$
Query	A BiTRDF knowledge store, a set of arguments	Process the arguments against the BiTRDF knowledge store	Bitemporal resources or bitemporal triples that satisfy the query
Output	A BiTRDF graph	Serialize the BiTRDF graph to Turtle syntax [19]	A text file that contains a BiTRDF graph written in Turtle syntax [19]

Appendix B. Running Example Stored in OOP+Dictionary

```

subject_dictionary: {
  :NJ[0,now][0,now]: {bitrdf:type[0,now][0,now]: {state[0,now][0,now]}},
  :NY[0,now][0,now]: {bitrdf:type[0,now][0,now]: {state[0,now][0,now]}},
  :CO[0,now][0,now]: {bitrdf:type[0,now][0,now]: {state[0,now][0,now]}},
  :NYC[0,now][0,now]: {bitrdf:type[0,now][0,now]: {city[0,now][0,now]},
    :locatesAt[0,now][0,now]: {NY[0,now][0,now]}},
  :CUNY[0,now][0,now]: {bitrdf:type[0,now][0,now]: {Univ[0,now][0,now]},
    :locatesAt[0,now][0,now]: {NY[0,now][0,now]}},
  :CU[0,now][0,now]: {bitrdf:type[0,now][0,now]: {Univ[0,now][0,now]},
    :locatesAt[0,now][0,now]: {CO[0,now][0,now]}},
  :Lu[1,now][1,now]: {bitrdf:type[1,now][1,now]: {person[0,now][0,now]},
    :livesIn[1,5][1,5]: {NJ[0,now][0,now]},
    :studyAt[1,now][1,now]: {CUNY[0,now][0,now]},
    :livesIn[5,10][5,10]: {NY[0,now][0,now]},
    :livesIn[10,now][10,now]: {CO[0,now][0,now]},
    :worksAt[10,now][10,now]: {CU[0,now][0,now]},
    :worksWith[10,now][10,now]: {Ab[1,now][1,now]}},
  :Ab[1,now][1,now]: {bitrdf:type[1,now][1,now]: {person[0,now][0,now]},
    :livesIn[5,10][5,10]: {NJ[0,now][0,now]},
    :livesIn[10,now][10,now]: {NY[0,now][0,now]}}
}
predicate_dictionary: {
  bitrdf:type[0,now][0,now]: {state[0,now][0,now]: {NJ[0,now][0,now],
    :NY[0,now][0,now],
    :CO[0,now][0,now]},
    :city[0,now][0,now]: {NYC[0,now][0,now]},
    :Univ[0,now][0,now]: {CUNY[0,now][0,now],
    :CU[0,now][0,now]}},
  :locatesAt[0,now][0,now]: {NY[0,now][0,now]: {NYC[0,now][0,now],
    :CUNY[0,now][0,now]},
    :CO[0,now][0,now]: {CU[0,now][0,now]}},
  bitrdf:type[1,now][1,now]: {person[0,now][0,now]: {Ab[1,now][1,now],
    :Lu[1,now][1,now]}},
  :livesIn[1,5][1,5]: {NJ[0,now][0,now]: {Lu[1,now][1,now]}},
  :studyAt[1,now][1,now]: {CUNY[0,now][0,now]: {Lu[1,now][1,now]}},
  :livesIn[5,10][5,10]: {NY[0,now][0,now]: {Lu[1,now][1,now]},
    :NJ[0,now][0,now]: {Ab[1,now][1,now]}},
}

```

```

: livesIn[10,now][10,now]: { :CO[0,now][0,now]: { :Lu[1,now][1,now]},
                        :NY[0,now][0,now]: { :Ab[1,now][1,now]}},
: worksAt[10,now][10,now]: { :CU[0,now][0,now]: { :Lu[1,now][1,now]}},
: worksWith[10,now][10,now]: { :Ab[1,now][1,now]: { :Lu[1,now][1,now]}},
}
object_dictionary: {
: state[0,now][0,now]: { :NJ[0,now][0,now]: {bitrdf:type[0,now][0,now]},
                        :NY[0,now][0,now]: {bitrdf:type[0,now][0,now]},
                        :CO[0,now][0,now]: {bitrdf:type[0,now][0,now]}},
: city[0,now][0,now]: { :NYC[0,now][0,now]: {bitrdf:type[0,now][0,now]}},
: Univ[0,now][0,now]: { :CUNY[0,now][0,now]: {bitrdf:type[0,now][0,now]},
                        :CU[0,now][0,now]: {bitrdf:type[0,now][0,now]}},
: NY[0,now][0,now]: { :NYC[0,now][0,now]: { :locatesAt[0,now][0,now]},
                        :CUNY[0,now][0,now]: { :locatesAt[0,now][0,now]},
                        :Lu[1,now][1,now]: { :livesIn[5,10][5,10]},
                        :Ab[1,now][1,now]: { :livesIn[10,now][10,now]}},
: CO[0,now][0,now]: { :CU[0,now][0,now]: { :locatesAt[0,now][0,now]},
                        :Lu[1,now][1,now]: { :livesIn[10,now][10,now]}},
: person[0,now][0,now]: { :Lu[1,now][1,now]: {bitrdf:type[1,now][1,now]},
                          :Ab[1,now][1,now]: {bitrdf:type[1,now][1,now]}},
: NJ[0,now][0,now]: { :Lu[1,now][1,now]: { :livesIn[1,5][1,5]},
                      :Ab[1,now][1,now]: { :livesIn[5,10][5,10]}},
: CUNY[0,now][0,now]: { :Lu[1,now][1,now]: { :studyAt[1,now][1,now]}},
: CU[0,now][0,now]: { :Lu[1,now][1,now]: { :worksAt[10,now][10,now]}},
: Ab[1,now][1,now]: { :Lu[1,now][1,now]: { :worksWith[10,now][10,now]}},
}

```

Appendix C. Running Example Stored in OOP+DataFrame

```

=====BiTemporal Triple Table=====
      S              P              O
0      :NJ      bitrdf:type[0,now][0,now]      :state
1      :NY      bitrdf:type[0,now][0,now]      :state
2      :CO      bitrdf:type[0,now][0,now]      :state
3      :NYC     bitrdf:type[0,now][0,now]      :city
4      :CUNY     bitrdf:type[0,now][0,now]      :Univ
5      :CU       bitrdf:type[0,now][0,now]      :Univ
6      :NYC      :locatesAt[0,now][0,now]      :NY
7      :CUNY     :locatesAt[0,now][0,now]      :NY
8      :CU       :locatesAt[0,now][0,now]      :CO
9      :Lu       bitrdf:type[1,now][1,now]      :person
10     :Ab       bitrdf:type[1,now][1,now]      :person
11     :Lu              :livesIn[1,5][1,5]      :NJ
12     :Lu              :studyAt[1,now][1,now]   :CUNY
13     :Lu              :livesIn[5,10][5,10]     :NY
14     :Ab              :livesIn[5,10][5,10]     :NJ
15     :Lu              :livesIn[10,now][10,now]  :CO
16     :Lu              :worksAt[10,now][10,now]  :CU
17     :Ab              :livesIn[10,now][10,now]  :NY
18     :Lu              :worksWith[10,now][10,now]:Ab

=====BiTemporal Entity Table=====
      Entity
0      :NJ[0,now][0,now]
1      :state[0,now][0,now]
2      :NY[0,now][0,now]
3      :CO[0,now][0,now]
4      :NYC[0,now][0,now]
5      :city[0,now][0,now]
6      :CUNY[0,now][0,now]
7      :Univ[0,now][0,now]
8      :CU[0,now][0,now]
9      :Lu[1,now][1,now]
10     :person[0,now][0,now]
11     :Ab[1,now][1,now]

=====BiTemporal Predicate Table=====
      Predicate
0      bitrdf:type[0,now][0,now]
6      :locatesAt[0,now][0,now]
9      bitrdf:type[1,now][1,now]
11     :livesIn[1,5][1,5]

```

```

12      :studyAt[1,now][1,now]
13      :livesIn[5,10][5,10]
15      :livesIn[10,now][10,now]
16      :worksAt[10,now][10,now]
18      :worksWith[10,now][10,now]

```

Appendix D. Running Example Stored in OOP+List

```

bitemporal_graph = [
:NY[0,now][0,now] bitrdf:type[0,now][0,now] :state[0,now][0,now] .,
:NY[0,now][0,now] bitrdf:type[0,now][0,now] :state[0,now][0,now] .,
:CO[0,now][0,now] bitrdf:type[0,now][0,now] :state[0,now][0,now] .,
:NYC[0,now][0,now] bitrdf:type[0,now][0,now] :city[0,now][0,now] .,
:CUNY[0,now][0,now] bitrdf:type[0,now][0,now] :Univ[0,now][0,now] .,
:CU[0,now][0,now] bitrdf:type[0,now][0,now] :Univ[0,now][0,now] .,
:NYC[0,now][0,now] :locatesAt[0,now][0,now] :NY[0,now][0,now] .,
:CUNY[0,now][0,now] :locatesAt[0,now][0,now] :NY[0,now][0,now] .,
:CU[0,now][0,now] :locatesAt[0,now][0,now] :CO[0,now][0,now] .,
:Lu[1,now][1,now] bitrdf:type[1,now][1,now] :person[0,now][0,now] .,
:Ab[1,now][1,now] bitrdf:type[1,now][1,now] :person[0,now][0,now] .,
:Lu[1,now][1,now] :livesIn[1,5][1,5] :NJ[0,now][0,now] .,
:Lu[1,now][1,now] :studyAt[1,now][1,now] :CUNY[0,now][0,now] .,
:Lu[1,now][1,now] :livesIn[5,10][5,10] :NY[0,now][0,now] .,
:Ab[1,now][1,now] :livesIn[5,10][5,10] :NJ[0,now][0,now] .,
:Lu[1,now][1,now] :livesIn[10,now][10,now] :CO[0,now][0,now] .,
:Lu[1,now][1,now] :worksAt[10,now][10,now] :CU[0,now][0,now] .,
:Ab[1,now][1,now] :livesIn[10,now][10,now] :NY[0,now][0,now] .,
:Lu[1,now][1,now] :worksWith[10,now][10,now] :Ab[1,now][1,now] .
]

```

Appendix E. Running Example Stored in Database+PostgreSQL

id	name	vtbegin	vtend	ttbegin	ttend	s	p	o
1	:NJ	0	now	0	now	1		
2	bitrdf:type	0	now	0	now		1	
3	:state	0	now	0	now			1
4	:NY	0	now	0	now	2		
5	bitrdf:type	0	now	0	now		2	
6	:state	0	now	0	now			2
7	:CO	0	now	0	now	3		
8	bitrdf:type	0	now	0	now		3	
9	:state	0	now	0	now			3
10	:NYC	0	now	0	now	4		
11	bitrdf:type	0	now	0	now		4	
12	:city	0	now	0	now			4
13	:CUNY	0	now	0	now	5		
14	bitrdf:type	0	now	0	now		5	
15	:Univ	0	now	0	now			5
16	:CU	0	now	0	now	6		
17	bitrdf:type	0	now	0	now		6	
18	:Univ	0	now	0	now			6
19	:NYC	0	now	0	now	7		
20	:locatesAt	0	now	0	now		7	
21	:NY	0	now	0	now			7
22	:CUNY	0	now	0	now	8		
23	:locatesAt	0	now	0	now		8	
24	:NY	0	now	0	now			8
25	:CU	0	now	0	now	9		
26	:locatesAt	0	now	0	now		9	
27	:CO	0	now	0	now			9
28	:Lu	1	now	1	now	10		
29	bitrdf:type	1	now	1	now		10	
30	:person	0	now	0	now			10
31	:Ab	1	now	1	now	11		
32	bitrdf:type	1	now	1	now		11	
33	:person	0	now	0	now			11
34	:Lu	1	now	1	now	12		
35	:livesIn	1	5	1	5		12	

36		:NJ		0		now		0		now				12
37		:Lu		1		now		1		now		13		
38		:studyAt		1		now		1		now				13
39		:CUNY		0		now		0		now				13
40		:Lu		1		now		1		now		14		
41		:livesIn		5		10		5		10				14
42		:NY		0		now		0		now				14
43		:Ab		1		now		1		now		15		
44		:livesIn		5		10		5		10				15
45		:NJ		0		now		0		now				15
46		:Lu		1		now		1		now		16		
47		:livesIn		10		now		10		now				16
48		:CO		0		now		0		now				16
49		:Lu		1		now		1		now		17		
50		:worksAt		10		now		10		now				17
51		:CU		0		now		0		now				17
52		:Ab		1		now		1		now		18		
53		:livesIn		10		now		10		now				18
54		:NY		0		now		0		now				18
55		:Lu		1		now		1		now		19		
56		:worksWith		10		now		10		now				19
57		:Ab		1		now		1		now				19
(57 rows)														

Appendix F. Running Example Stored in Database+Dictionary

```
[
{'s_r': ':NJ', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':state', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':NY', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':state', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':CO', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':state', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':NYC', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':city', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':CUNY', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':Univ', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':CU', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':Univ', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':NYC', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': ':locatesAt', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':NY', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':CUNY', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': ':locatesAt', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':NY', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':CU', 'svts': '0', 'svte': 'now', 'stts': '0', 'stte': 'now',
 'p_r': ':locatesAt', 'pvts': '0', 'pvte': 'now', 'ptts': '0', 'ptte': 'now',
 'o_r': ':CO', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':Lu', 'svts': '1', 'svte': 'now', 'stts': '1', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '1', 'pvte': 'now', 'ptts': '1', 'ptte': 'now',
 'o_r': ':person', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':Ab', 'svts': '1', 'svte': 'now', 'stts': '1', 'stte': 'now',
 'p_r': 'bitrdf:type', 'pvts': '1', 'pvte': 'now', 'ptts': '1', 'ptte': 'now',
 'o_r': ':person', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':Lu', 'svts': '1', 'svte': 'now', 'stts': '1', 'stte': 'now',
 'p_r': ':livesIn', 'pvts': '1', 'pvte': '5', 'ptts': '1', 'ptte': '5',
 'o_r': ':NJ', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':Lu', 'svts': '1', 'svte': 'now', 'stts': '1', 'stte': 'now',
 'p_r': ':studyAt', 'pvts': '1', 'pvte': 'now', 'ptts': '1', 'ptte': 'now',
 'o_r': ':CUNY', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':Lu', 'svts': '1', 'svte': 'now', 'stts': '1', 'stte': 'now',
 'p_r': ':livesIn', 'pvts': '5', 'pvte': '10', 'ptts': '5', 'ptte': '10',
 'o_r': ':NY', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
{'s_r': ':Ab', 'svts': '1', 'svte': 'now', 'stts': '1', 'stte': 'now',
 'p_r': ':livesIn', 'pvts': '5', 'pvte': '10', 'ptts': '5', 'ptte': '10',
 'o_r': ':NJ', 'ovts': '0', 'ovte': 'now', 'otts': '0', 'otte': 'now'},
]
```

```
{'s_r':'Lu','svts':'1','svte':'now','stts':'1','stte':'now',
  'p_r':'livesIn','pvts':'10','pvte':'now','ptts':'10','ptte':'now',
  'o_r':'CO','ovts':'0','ovte':'now','otts':'0','otte':'now'},
{'s_r':'Lu','svts':'1','svte':'now','stts':'1','stte':'now',
  'p_r':'worksAt','pvts':'10','pvte':'now','ptts':'10','ptte':'now',
  'o_r':'CU','ovts':'0','ovte':'now','otts':'0','otte':'now'},
{'s_r':'Ab','svts':'1','svte':'now','stts':'1','stte':'now',
  'p_r':'livesIn','pvts':'10','pvte':'now','ptts':'10','ptte':'now',
  'o_r':'NY','ovts':'0','ovte':'now','otts':'0','otte':'now'},
{'s_r':'Lu','svts':'1','svte':'now','stts':'1','stte':'now',
  'p_r':'worksWith','pvts':'10','pvte':'now','ptts':'10','ptte':'now',
  'o_r':'Ab','ovts':'1','ovte':'now','otts':'1','otte':'now'}
]
```

Appendix G. Running Example Stored in Database+List

```
[
[':NJ','0','now','0','now','bitrdf:type','0','now','0','now',
  ':state','0','now','0','now'],
[':NY','0','now','0','now','bitrdf:type','0','now','0','now',
  ':state','0','now','0','now'],
[':CO','0','now','0','now','bitrdf:type','0','now','0','now',
  ':state','0','now','0','now'],
[':NYC','0','now','0','now','bitrdf:type','0','now','0','now',
  ':city','0','now','0','now'],
[':CUNY','0','now','0','now','bitrdf:type','0','now','0','now',
  ':Univ','0','now','0','now'],
[':CU','0','now','0','now','bitrdf:type','0','now','0','now',
  ':Univ','0','now','0','now'],
[':NYC','0','now','0','now','locatesAt','0','now','0','now',
  ':NY','0','now','0','now'],
[':CUNY','0','now','0','now','locatesAt','0','now','0','now',
  ':NY','0','now','0','now'],
[':CU','0','now','0','now','locatesAt','0','now','0','now',
  ':CO','0','now','0','now'],
[':Lu','1','now','1','now','bitrdf:type','1','now','1','now',
  ':person','0','now','0','now'],
[':Ab','1','now','1','now','bitrdf:type','1','now','1','now',
  ':person','0','now','0','now'],
[':Lu','1','now','1','now','livesIn','1','5','1','5',
  ':NJ','0','now','0','now'],
[':Lu','1','now','1','now','studyAt','1','now','1','now',
  ':CUNY','0','now','0','now'],
[':Lu','1','now','1','now','livesIn','5','10','5','10',
  ':NY','0','now','0','now'],
[':Ab','1','now','1','now','livesIn','5','10','5','10',
  ':NJ','0','now','0','now'],
[':Lu','1','now','1','now','livesIn','10','now','10','now',
  ':CO','0','now','0','now'],
[':Lu','1','now','1','now','worksAt','10','now','10','now',
  ':CU','0','now','0','now'],
[':Ab','1','now','1','now','livesIn','10','now','10','now',
  ':NY','0','now','0','now'],
[':Lu','1','now','1','now','worksWith','10','now','10','now',
  ':Ab','1','now','1','now']
]
```

Appendix H. BiTemporal Query Examples

Example A1. Select all subjects whose resource is “:Lu”. The result is shown in Table A2.

Table A2. Query Output: Select all subjects whose resource is “:Lu”.

Subject
:Lu[1,now][1,now]

Example A2. Select all predicates whose resource is “:livesIn”. The result is shown in Table A3.

Table A3. Query Output: Select all predicates whose resource is “:livesIn”.

Predicate
:livesIn[1,5][1,5)
:livesIn[5,10)[5,10)
:livesIn[10,now][10,now]

Example A3. Select all objects whose resource is “:NY”. The result is shown in Table A4.

Table A4. Query Output: Select all objects whose resource is “:NY”.

Predicate
:NY[0,now][0,now]

Example A4. Select all bitemporal triples in the sample store. The result is shown in Table A5.

Table A5. Query Output: Select all bitemporal triples in the sample store.

Subject	Predicate	Object
:NJ[0,now][0,now]	bitrdf:type[0,now][0,now]	:state[0,now][0,now]
:NY[0,now][0,now]	bitrdf:type[0,now][0,now]	:state[0,now][0,now]
:CO[0,now][0,now]	bitrdf:type[0,now][0,now]	:state[0,now][0,now]
:NYC[0,now][0,now]	bitrdf:type[0,now][0,now]	:city[0,now][0,now]
:NYC[0,now][0,now]	:locatesAt[0,now][0,now]	:NY[0,now][0,now]
:CUNY[0,now][0,now]	bitrdf:type[0,now][0,now]	:Univ[0,now][0,now]
:CUNY[0,now][0,now]	:locatesAt[0,now][0,now]	:NY[0,now][0,now]
:CU[0,now][0,now]	bitrdf:type[0,now][0,now]	:Univ[0,now][0,now]
:CU[0,now][0,now]	:locatesAt[0,now][0,now]	:CO[0,now][0,now]
:Lu[1,now][1,now]	bitrdf:type[1,now][1,now]	:person[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[1,5)[1,5)	:NJ[0,now][0,now]
:Lu[1,now][1,now]	:studyAt[1,now][1,now]	:CUNY[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[5,10)[5,10)	:NY[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[10,now][10,now]	:CO[0,now][0,now]
:Lu[1,now][1,now]	:worksAt[10,now][10,now]	:CU[0,now][0,now]
:Lu[1,now][1,now]	:worksWith[10,now][10,now]	:Ab[1,now][1,now]
:Ab[1,now][1,now]	bitrdf:type[1,now][1,now]	:person[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[5,10)[5,10)	:NJ[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[10,now][10,now]	:NY[0,now][0,now]

Example A5. Select all bitemporal triples whose subjects’ resource is “:Lu”. The result is shown in Table A6.

Table A6. Query Output: Select all bitemporal triples whose subjects' resource is ":Lu".

Subject	Predicate	Object
:Lu[1,now][1,now]	bitrdf:type[1,now][1,now]	:person[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[1,5][1,5)	:NJ[0,now][0,now]
:Lu[1,now][1,now]	:studyAt[1,now][1,now]	:CUNY[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[5,10)[5,10)	:NY[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[10,now][10,now]	:CO[0,now][0,now]
:Lu[1,now][1,now]	:worksAt[10,now][10,now]	:CU[0,now][0,now]
:Lu[1,now][1,now]	:worksWith[10,now][10,now]	:Ab[1,now][1,now]

Example A6. Select all bitemporal triples whose predicates' resource is ":livesIn". The result is shown in Table A7.

Table A7. Query Output: Select all bitemporal triples whose predicates' resource is ":livesIn".

Subject	Predicate	Object
:Lu[1,now][1,now]	:livesIn[1,5)(1,5)	:NJ[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[5,10)[5,10)	:NY[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[5,10)[5,10)	:NJ[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[10,now][10,now]	:CO[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[10,now][10,now]	:NY[0,now][0,now]

Example A7. Select all bitemporal triples whose objects' resource is ":NY". The result is shown in Table A8.

Table A8. Query Output: Select all bitemporal triples whose objects' resource is ":NY".

Subject	Predicate	Object
:NYC[0,now][0,now]	:locatesAt[0,now][0,now]	:NY[0,now][0,now]
:CUNY[0,now][0,now]	:locatesAt[0,now][0,now]	:NY[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[5,10)[5,10)	:NY[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[10,now][10,now]	:NY[0,now][0,now]

If a valid time constraint v_t is specified in a query, all result will be filtered with a constraint of the valid time v_t . If AS OF clause is not specified, the default *now* will be used.

Example A8. Select all bitemporal triples whose predicates' resource is ":livesIn" and are valid at time 1. The result is shown in Table A9.

Table A9. Query Output: Select all bitemporal triples whose predicates' resource is ":livesIn" and are valid at time 1.

Subject	Predicate	Object
:Lu[1,now][1,now]	:livesIn[1,5)(1,5)	:NJ[0,now][0,now]

Example A9. Select all bitemporal triples whose predicates' resource is ":livesIn" and are valid at time 5. The result is shown in Table A10.

Table A10. Query Output: Select all bitemporal triples whose predicates' resource is ":livesIn" and are valid at time 5.

Subject	Predicate	Object
:Lu[1,now][1,now]	:livesIn[5,10][5,10)	:NY[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[5,10][5,10)	:NJ[0,now][0,now]

Example A10. Select all bitemporal triples whose predicates' resource is ":livesIn" and are valid at time 10. The result is shown in Table A11.

Table A11. Query Output: Select all bitemporal triples whose predicates' resource is ":livesIn" and are valid at time 10.

Subject	Predicate	Object
:Lu[1,now][1,now]	:livesIn[10,now][10,now]	:CO[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[10,now][10,now]	:NY[0,now][0,now]

If AS OF clause specifies a transaction time instant t_t , the knowledge store will be rolled back to specified transaction time instant t_t , and query as of the knowledge store were at that time t_t .

Example A11. Select all bitemporal triples whose predicates' resource is ":livesIn" as of time 10. The result is shown in Table A12.

Table A12. Query Output: Select all bitemporal triples whose predicates' resource is ":livesIn" as of time 10.

Subject	Predicate	Object
:Lu[1,now][1,now]	:livesIn[1,5)[1,5)	:NJ[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[5,10)[5,10)	:NY[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[5,10)[5,10)	:NJ[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[10,now][10,now]	:CO[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[10,now][10,now]	:NY[0,now][0,now]

Example A12. Select all bitemporal triples whose predicates' resource is ":livesIn" as of time 5. The result is shown in Table A13.

Table A13. Query Output: Select all bitemporal triples whose predicates' resource is ":livesIn" as of time 5.

Subject	Predicate	Object
:Lu[1,now][1,now]	:livesIn[1,5)[1,5)	:NJ[0,now][0,now]
:Lu[1,now][1,now]	:livesIn[5,10)[5,10)	:NY[0,now][0,now]
:Ab[1,now][1,now]	:livesIn[5,10)[5,10)	:NJ[0,now][0,now]

Appendix I. Experiment Results in Detail

We summarize the results of memory consumption, loading time, and query time in following tables.

Table A14. Summary of Memory Consumption (MB).

Alternative/DataSet	D1	D2	D3	D4	D5	D6
OOP + Dictionary	831	1247	1468	1330	536	554
OOP + DataFrame	1101	1260	1429	1329	1094	658
OOP + List	1041	1303	1814	1525	1512	1334
DB + PostgreSQL	1667	3401	2323	456	1983	2532
DB + Dictionary	853	1701	3256	5054	3404	2479
DB + List	638	1273	2526	4640	3115	2779

Table A15. Summary of Loading Time (seconds).

Alternative/DataSet	D1	D2	D3	D4	D5	D6
OOP + Dictionary	12.87	26.67	68.10	335.50	1069.00	2849.00
OOP + DataFrame	14.75	30.60	62.13	260.33	773.40	2523.10
OOP + List	9.14	18.31	36.50	76.49	406.68	1296.00
DB + PostgreSQL	5.70	15.00	35.00	102.00	254.00	598.00
DB + Dictionary	2.24	4.37	8.94	17.50	37.27	91.90
DB + List	2.85	6.13	12.79	26.57	107.06	409.38

Table A16. Summary of Query Time (seconds).

Alternative/DataSet	D1	D2	D3	D4	D5	D6
OOP + Dictionary	0.13	0.33	2.90	8.50	25.00	67.00
OOP + DataFrame	0.25	0.40	0.87	3.67	8.55	24.90
OOP + List	0.11	0.09	0.45	2.51	10.32	22.00
DB + PostgreSQL	0.12	0.18	0.25	1.37	5.68	15.25
DB + Dictionary	0.07	0.16	0.34	3.17	10.20	30.10
DB + List	0.05	0.10	0.21	0.43	5.94	18.60

References

1. Tim, B.-L. Enabling Standards & Technologies—Layer Cake. Available online: <https://www.w3.org/2002/Talks/04-sweb/slide12-0.html> (accessed on 10 June 2022).
2. Wu, D. BiTRDF: Extending RDF for Bitemporal Data. Ph.D. Thesis, The Graduate Center, City University of New York, New York, NY, USA, 2022.
3. Tansel, A.U.; Wu, D.; Wang, H.T. Time Travel with the BiTemporal RDF Model. *Mathematics* **2025**, *13*, 2109. [\[CrossRef\]](#)
4. Wu, D.; Wang, H.T.; Tansel, A.U. A survey for managing temporal data in RDF. *Inf. Systems* **2024**, *122*, 102368. [\[CrossRef\]](#)
5. Carroll, J.J.; Bizer, C.; Hayes, P.; Stickler, P. Named Graphs, Provenance and Trust. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*; Association for Computing Machinery: New York, NY, USA, 2005; pp. 613–622. [\[CrossRef\]](#)
6. Gutierrez, C.; Hurtado, C.A.; Vaisman, A. Temporal RDF. In *Proceedings of the Second European Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 93–107.
7. Gutierrez, C.; Hurtado, C.A.; Vaisman, A. Introducing Time into RDF. *IEEE Trans. Knowl. Data Eng.* **2007**, *19*, 207–218. [\[CrossRef\]](#)
8. Nguyen, V.; Bodenreider, O.; Sheth, A. Don't Like RDF Reification? Making Statements About Statements Using Singleton Property. In *WWW '14: Proceedings of the 23rd International Conference on World Wide Web*; Association for Computing Machinery: New York, NY, USA, 2014; pp. 759–770. [\[CrossRef\]](#)
9. Welty, C.; Fikes, R.; Makarios, S. A reusable ontology for fluents in OWL. In *FOIS*; IOS Press: Amsterdam, The Netherlands, 2006; pp. 226–236.
10. Broekstra, J.; Kampman, A.; van Harmelen, F. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the International Semantic Web Conference (ISWC)*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 54–68. [\[CrossRef\]](#)

11. Ali, W.; Saleem, M.; Yao, B.; Hogan, A.; Ngonga Ngomo, A.C. A Survey of RDF Stores & SPARQL Engines for Querying Knowledge Graphs. *VLDB J.* **2022**, *31*, 1–26. [CrossRef]
12. Weiss, C.; Karras, P.; Bernstein, A. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proc. VLDB Endow.* **2008**, *1*, 1008–1019. [CrossRef]
13. Neumann, T.; Weikum, G. RDF-3X: A RISC-Style Engine for RDF. *VLDB Endow.* **2008**, *1*, 647–659. [CrossRef]
14. Cuevas, I.; Hogan, A. Versioned Queries over RDF Archives: All You Need is SPARQL? In Proceedings of the MEPDaW@ISWC, Virtual Event, 1 November 2020; pp. 43–52. Available online: <https://mepdaw-ws.github.io/2020/> (accessed on 11 April 2026)
15. Taelman, R.; Vander Sande, M.; Van Herwegen, J.; Mannens, E.; Verborgh, R. Triple Storage for Random-Access Versioned Querying of RDF Archives. *J. Web Semant.* **2019**, *54*, 4–28. [CrossRef]
16. Snodgrass, R.T. *The TSQL2 Temporal Query Language*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012.
17. The PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Relational Database. Available online: <https://www.postgresql.org> (accessed on 10 January 2022).
18. Tansel, A.U. Efficient Management of Temporal Knowledge. U.S. Patent 10055450, 21 August 2018.
19. Beckett, D.; Berners-Lee, T.; Prudhommeaux, E.; Carothers, G. RDF 1.1 Turtle. *W3C Recomm.* **2014**, *25*, 18–31.
20. Duan, S.; Kementsietsidis, A.; Srinivas, K.; Udrea, O. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*; SIGMOD: Athens, Greece, 2011; pp. 145–156.
21. Beauregard, B.; Das, S.; Perry, M.; Wu, Z. Oracle Spatial and Graph: Benchmarking a Trillion Edges RDF Graph. 2016. Available online: <https://api.semanticscholar.org/CorpusID:14460982> (accessed on 1 June 2020)
22. Perry, M.; Estrada, A.; Das, S.; Banerjee, J. Developing GeoSPARQL Applications with Oracle Spatial and Graph. In *Proceedings of the SSN-TC/OrdRing@ISWC*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 57–61.
23. Fernandes, D.; Bernardino, J. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. *Data* **2018**, *18*, 373–380.
24. Cerans, K.; Barzdins, G.; Liepins, R.; Ovcinnikova, J.; Rikacovs, S.; Sprogis, A. Graphical Schema Editing for Stardog OWL/RDF Databases using OWLGrEd/S. In *Proceedings of the OWLED*; CEUR-WS.org: Aachen, Germany, 2012; Volume 849.
25. Pauwels, P.; De Farias, T.M.; Zhang, C.; Roxin, A.; Beetz, J.; De Roo, J.; Nicolle, C. Querying and reasoning over large scale building data sets: An outline of a performance benchmark. In *Proceedings of the International Workshop on Semantic Big Data*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 1–6.
26. Taelman, R.; Vander Sande, M.; Verborgh, R. Bridges between GraphQL and RDF. In *W3C Workshop on Web Standardization for Graph Data*; W3C: Cambridge, MA, USA, 2019. Available online: <https://www.w3.org/Data/events/data-ws-2019/assets/position/Ruben%20Taelman.pdf> (accessed on 11 April 2026)
27. Erling, O. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* **2012**, *35*, 3–8.
28. Erling, O.; Mikhailov, I. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 7–24.
29. Erling, O.; Mikhailov, I. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 501–519.
30. Faralli, S.; Velardi, P.; Yusifli, F. Multiple knowledge GraphDB (MKGDB). In *Proceedings of the 12th Language Resources and Evaluation Conference*; European Language Resources Association: Paris, France, 2020; pp. 2325–2331.
31. Güting, R.H. GraphDB: Modeling and querying graphs in databases. In *Proceedings of the VLDB*; Citeseer: Santiago, Chile, 1994; Volume 94, pp. 12–15.
32. Atemezing, G.A.; Amardeilh, F. Benchmarking commercial RDF stores with publications office dataset. In *Proceedings of the European Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 379–394.
33. Khadilkar, V.; Kantarcioglu, M.; Thuraishingham, B.; Castagna, P. Jena-HBase: A distributed, scalable and efficient RDF triple store. In *Proceedings of the 11th International Semantic Web Conference Posters & Demonstrations Track, ISWC-PD*; Citeseer: Santiago, Chile, 2012; Volume 12, pp. 85–88.
34. Morsey, M.; Lehmann, J.; Auer, S.; Ngonga Ngomo, A.C. DBpedia SPARQL benchmark—performance assessment with real queries on real data. In *Proceedings of the International Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 454–469.
35. Owens, A.; Seaborne, A.; Gibbins, N. *Clustered TDB: A Clustered Triple Store for Jena*; University of Southampton: Southampton, UK, 2008.
36. Le-Tuan, A.; Hayes, C.; Wylot, M.; Le-Phuoc, D. RDF4Led: An RDF engine for lightweight edge devices. In *Proceedings of the 8th International Conference on the Internet of Things*; Association for Computing Machinery: New York, NY, USA, 2018; pp. 1–8.
37. Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; Banerjee, J. RDFox: A highly-scalable RDF store. In *Proceedings of the International Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 3–20.

38. Krech, D. Rdfliib: A Python Library for Working with RDF. 2006. Available online: <https://github.com/RDFLib/rdfliib> (accessed on 1 June 2020).
39. Harris, S.; Gibbins, N. 3store: Efficient bulk RDF Storage. In Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), Sanibel Island, FL, USA, 20 October 2003.
40. Harris, S.; Lamb, N.; Shadbolt, N. 4store: The design and implementation of a clustered RDF store. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*; University of Oxford: Oxford, UK, 2009; Volume 94.
41. Oliveira, F.J.M.; Ramalho, J.C. Open web ontobud: An open source RDF4J frontend. In *Proceedings of the 9th Symposium on Languages, Applications and Technologies (SLATE 2020)*; Schloss Dagstuhl–Leibniz-Zentrum für Informatik: Wadern, Germany, 2020.
42. Punnoose, R.; Crainiceanu, A.; Rapp, D. Rya: A scalable RDF triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*; Association for Computing Machinery: New York, NY, USA, 2012; pp. 1–8.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.