QuaQue: Design and SQL Implementation of Condensed Algebra for Concurrent Versioning of Knowledge Graphs

Jey Puget Gil, Emmanuel Coquery, John Samuel, Gilles Gesquière
Universite Claude Bernard Lyon 1, CNRS, INSA Lyon, Université Lumière Lyon 2,
Ecole Centrale de Lyon, CPE Lyon, LIRIS, UMR 5205
Villeurbanne, France
e-mail: {jey.puget-gil | emmanuel.coquery | john.samuel | gilles.gesquiere}@liris.cnrs.fr

***Abstract*—The management of versioned knowledge graphs presents significant challenges, particularly in querying data across multiple versions efficiently. This paper introduces QuaQue, a key component of the ConVer-G system, which addresses this challenge by translating SPARQL (SPARQL Protocol and RDF Query Language) queries into SQL (Structured Query Language). QuaQue leverages a novel condensed algebra to operate on a relational model where versioning information is compactly stored using bitstrings. This approach allows for efficient querying of concurrent versions of knowledge graphs within a standard relational database system. We present the key concepts of our condensed algebra, detail the translation process from SPARQL algebra to SQL, and provide a comparative benchmark against a native RDF (Resource Description Framework) triple store, demonstrating the viability and performance benefits of our approach.**

Keywords-SQL; algebra; translation; concurrent versioning; relational.

I. Introduction

The representation of complex, evolving information has driven the widespread adoption of Knowledge Graphs (KGs) in both industry and academia [1]. However, as KGs move from static repositories to dynamic assets, there is a critical need for robust *concurrent versioning systems*. In domains, such as urban planning—where datasets undergo parallel modifications by multiple stakeholders—a linear history is insufficient. Systems must support branching, merging, and the analysis of concurrent states without excessive data redundancy.

While the Resource Description Framework and SPARQL are the de facto standards for KGs, they were primarily designed for static or monotonically increasing datasets. Native support for versioning remains a challenge. Standard approaches often rely on Named Graphs [2] to isolate versions. While accessing a single version remains efficient, this strategy leads to significant data duplication and increased query latency when querying across multiple versions. Consequently, efficiently querying across multiple versions remains an open problem in database research.

The ConVer-G project [3] addresses this by bridging the gap between graph versioning requirements and the mature optimization capabilities of Relational Database Management Systems (RDBMSs). At the core of ConVer-G is **QuaQue**, a system that translates SPARQL queries into SQL that exploits efficient bitwise operations for version filtering.

QuaQue leverages a *condensed relational model*. In RDF, a *quad* is a tuple that extends the standard triple with a graph identifier. We associate every quad with a *bitstring*, where each bit represents the validity of the quad in a specific version. This allows us to push version-filtering logic down to the RDBMS engine using efficient bitwise operations, significantly reducing the I/O overhead typically associated with multi-version queries.

This paper details the design and implementation of QuaQue. We introduce a *condensed algebra*—an extension of relational algebra tailored for bitstring-annotated relations—and define its translation to SQL. This enables the execution of complex graph pattern matching on standard PostgreSQL instances.

Our specific contributions are:

- A novel **condensed relational model** utilizing bitstrings for the storage of concurrent KG versions.
- The implementation of a **Condensed Algebra** and the QuaQue translator which maps SPARQL algebra to SQL.
- A comparative benchmark demonstrating that QuaQue outperforms a native RDF triple store (Apache Jena) in multi-version query scenarios.
- A fully reproducible approach, with the ConVer-G tool and the benchmark framework publicly available as open-source software.

The remainder of this paper is organized as follows. Section 2 presents the state of the art. Section 3 details the design of the QuaQue system and the condensed relational model. Section 4 presents the experimental evaluation and benchmark results. Finally, Section 5 concludes the paper and outlines future work.

II. State of the Art

The challenge of querying versioned data has been a long-standing topic in database research [4]. The relational model [5], introduced by Codd, provides a foundation with relational algebra and calculus as powerful query languages. The expressive power of these languages has been a central theme of research, with extensions proposed to handle more complex queries, such as those involving recursion [6][7] and aggregate functions [8]. A key milestone in bridging the gap between high-level query languages and efficient execution is the work of Ullman [9], which systematically addresses the translation of relational algebra and calculus queries into implementations, laying the groundwork for query processing and optimization. This connection is important, as the development of new algebras or extensions—such as those needed for versioned

or condensed data—must ultimately be supported by practical translation and execution strategies. Recent surveys, such as Hofer et al. [10], Jiang et al. [11][12], and Ji et al. [13] provide a comprehensive overview of the current state and challenges in knowledge graph construction, highlighting the increasing complexity of managing evolving and versioned knowledge graphs. They emphasize the need for scalable and efficient methods for both constructing and maintaining knowledge graphs, especially as these graphs become larger and more dynamic. This underscores the importance of advanced versioning and querying mechanisms to support the evolving requirements of knowledge graph applications. A notable example of leveraging versioned knowledge graphs in practice is the work by Gonzalez-Hevia and Gayo-Avello [14], who utilize Wikidata's edit history for knowledge graph refinement tasks. Their approach demonstrates the value of exploiting historical edit information to improve the quality and reliability of knowledge graphs, further motivating the need for efficient storage and querying of versioned data.

Recent work by Zhong et al. [15] further illustrates the importance of knowledge graphs in real-world applications, specifically in the domain of intelligent audit. Their study discusses both the opportunities and challenges of applying knowledge graphs to extract insights from complex, evolving data sources. This highlights the direct link between the need for advanced versioning and querying mechanisms—such as those discussed in this section—and the practical requirements of domains where data evolution and efficient cross-version analysis are critical for generating reliable insights.

A. Extending Relational Algebra

Relational algebra has been extended with techniques, such as rewriting queries with arbitrary aggregation functions using views, as discussed by Cohen et al. [16]. The extensibility of relational algebra has also been a subject of research, particularly in the context of supporting new data types and operations. Haas et al. [17] proposed an extensible query processor architecture that allows the integration of user-defined types and functions into the relational algebra framework. This extensibility adapts relational systems to various application domains, enabling the seamless incorporation of specialized operators and data structures without sacrificing the benefits of a declarative query language.

B. Relational Algebra in Query Languages

The principles of relational algebra have been foundational to the design of numerous query languages. SQL, the de facto standard for relational databases, is based on a tuple relational calculus, which is equivalent in expressive power to relational algebra [18]. The translation of SPARQL [19], the standard query language for RDF, to SQL has been a topic of interest for leveraging the performance and scalability of relational databases for semantic web data. Several systems, such as Ontop [20], have explored this translation, often relying on mappings between RDF and relational schemas.

Relational algebras are also applied outside the context of traditional databases. In qualitative spatial and temporal reasoning [21][22], relation algebras serve as formal tools for modeling and inferring relationships between spatial or temporal entities. For instance, Allen's interval algebra [23] provides a calculus for reasoning about temporal intervals, while the Region Connection Calculus (RCC) [24] is used for spatial reasoning. These algebras provide a formal, equational framework for deriving new knowledge from a set of base relations.

C. Semantic Versioned Querying: The Fundamentals

A contribution to the field of versioned querying for knowledge graphs is presented by Taelman et al. [25]. Their work systematically investigates the requirements and challenges of querying versioned semantic data, focusing on the formalization of versioned query semantics and the practical implications for query processing.

Taelman et al. introduce a formal framework for semantic versioned querying, distinguishing between different types of versioned queries, such as snapshot queries (retrieving data as it existed at a specific version), longitudinal queries (tracking the evolution of data across versions), and difference queries (identifying changes between versions). They emphasize the importance of clearly defined semantics for each query type, as ambiguity can lead to inconsistent or unintuitive results.

A key insight from their work is the need for query languages and systems to natively support version-aware operations, rather than treating versioning as an afterthought or external feature. They propose extensions to SPARQL that allow users to specify version constraints directly within queries, enabling more expressive and precise retrieval of historical or evolving data.

Overall, the work of Taelman et al. provides a foundational perspective on the semantics of querying versioned knowledge graphs. Their formalization of query types has been influential in our work, serving as a guideline for the capabilities our system aims to support.

D. Versioning Models for Evolving Data

Foundational models for data evolution stem from Software Configuration Management (SCM) [26], which distinguishes between sequential revisions and parallel variants. This distinction applies to knowledge graph versioning, where evolution involves both temporal changes and concurrent viewpoints [27]. SCM also differentiates between state-based (snapshots) and change-based (deltas) models [26]. In Model-Driven Engineering (MDE), these concepts extend to graph structures, where revisions are defined as sequences of atomic graph modifications [28]. Formalizing changes as graph operations enables structured reasoning about conflicts and merging [29], providing a theoretical basis for versioned query algebras.

Several strategies for storing versioned RDF data have emerged, balancing storage efficiency and query performance:

1) Independent Copies (IC): The IC or snapshot approach stores each version as a full copy [30]. While efficient for single-version querying (Version Materialization), it suffers from high storage redundancy.

2) Change-Based (CB): CB or delta-based versioning stores a base version and subsequent changes. This is space-efficient but requires costly reconstruction for querying. Tools, such as R43ples [29], R&WBase [31], and Stardog, follow this paradigm.

3) Timestamp-Based (TB): TB approaches annotate triples with validity intervals, facilitating "time-travel" queries. While suited for linear evolution, supporting branching adds complexity. Tools adopting this strategy include ConVer-G [32], Drydra [33], RDF-TX [34], v-RDFCSA [35], and x-RDF-3X [36], often drawing on temporal database concepts [37].

4) Fragment-Based (FB): FB or hybrid approaches partition the graph into independently versioned fragments, balancing storage and query performance. However, managing dependencies between fragments is complex. QuitStore [38] implements this by versioning modified files in a Git repository.

5) Graph Compression: Graph compression techniques, such as HDT (Header, Dictionary, Triples) [39], offer another perspective on efficient RDF storage. HDT is a binary format that achieves high compression ratios by utilizing dictionary encoding for RDF terms and a compact bitmap-based structure for the graph topology. Crucially, HDT files are designed to be queryable directly without decompression, providing excellent read performance for static datasets. However, the primary limitation of HDT and similar compression-centric approaches in the context of versioning is their static nature. They are optimized for read-only scenarios; modifying the data typically requires a computationally expensive reconstruction of the entire file. While one could theoretically store each version as a separate HDT file (effectively an optimized IC approach), this does not inherently solve the redundancy problem for small incremental changes, nor does it facilitate efficient cross-version querying or branching/merging operations. Therefore, while compression is a valuable component of storage optimization, it does not by itself constitute a complete versioning strategy for dynamic, evolving knowledge graphs.

6) Limitation: Git-based solutions, such as R&WBase [31] and QuitStore, require explicit checkout to query a version, hindering concurrent cross-version analysis [3]. Existing strategies struggle to balance storage efficiency and query performance for such analysis.

E. Summary: The Case for a Condensed TB Representation and Algebra

Current versioning strategies face a trade-off between storage and query efficiency, often lacking support for concurrent cross-version analysis [40]. A **condensed TB representation** addresses this by storing unique quads annotated with version validity, as seen in ConVer-G [3]. To exploit this model, a condensed algebra is needed to define operators on version-annotated structures, similar to specialized algebras, such as Knowledgebra [41]. Finally, an algebra-to-SQL translator (QuaQue) bridges the gap to efficient execution on relational systems [27].

III. Design

The QuaQue component of the ConVer-G system is a SPARQL-to-SQL translator designed to query a condensed relational model of versioned RDF data. Our approach is motivated by the need for efficient cross-version queries, which are cumbersome and inefficient with traditional triple-store-based versioning methods that replicate data for each version.

A. Condensed Relational Model

Our condensed model represents versioned RDF data in a relational database management system. We chose PostgreSQL for our implementation because it provides native support for bit string data types and efficient bitwise operations. This representation avoids storing duplicate quads that exist across multiple versions. The interaction between the SPARQL translation and this model is depicted in Figure 2. Versioned quad table, which stores quads (subject, predicate, object, named graph) along with a validity bitstring. Each bit in the validity string corresponds to a specific version, and a '1' at a given position indicates that the quad is present in that version. The structure of our condensed relational model is illustrated in Figure 1 and described as follows:

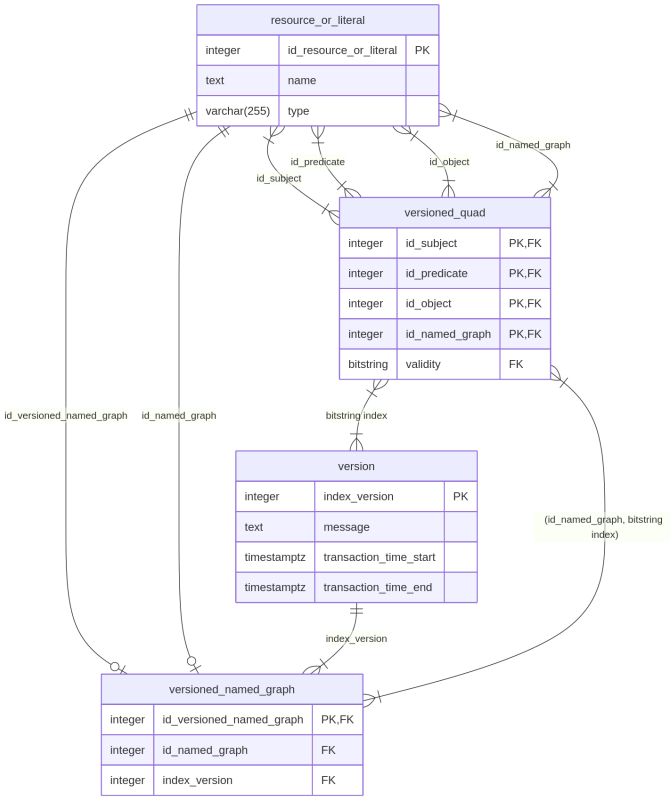


Figure 1. Relational model of the condensed representation for versioned RDF data.

The schema comprises the following relations:

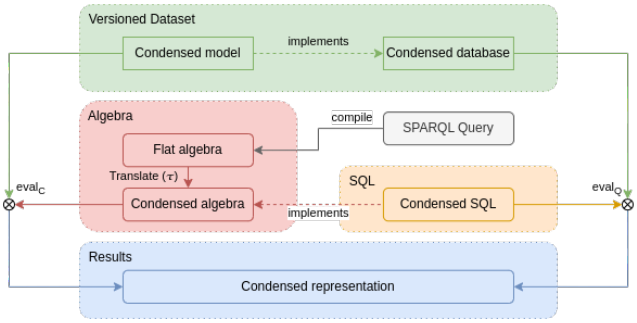


Figure 2. Overview of the SPARQL to SQL translation process in QuaQue.

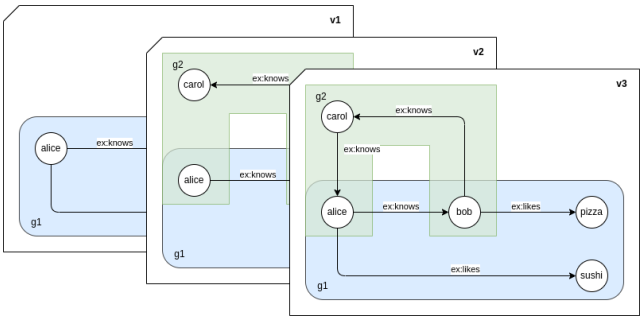


Figure 3. Graphical representation of the sample versioned RDF dataset.

- `versioned_quad`: This is the central table, storing unique quads (subject, predicate, object, named graph) as integer identifiers. Each entry includes a validity bitstring that indicates the versions in which the quad is present. This design minimizes redundancy by storing each unique quad only once, regardless of how many versions it appears in. An example is provided in Table III.
- `resource_or_literal`: This table acts as a dictionary, mapping RDF terms (URIs and literals) to the integer identifiers used in other tables. This practice, known as dictionary encoding, optimizes storage and join performance by replacing long strings with compact integers. See Table XI in the Appendix for an example.
- `version`: This table holds metadata specific to each version, such as its creation timestamp or a descriptive label. Separating version metadata allows for efficient retrieval of version-specific information without scanning the quad data.
- `versioned_named_graph`: This table links named graphs to the versions they belong to. This separates the association between graphs and versions from the quad data, adhering to database normalization principles to support scenarios where named graphs evolve independently across versions.
- `metadata`: This table stores additional metadata, which can be user-defined, about versions and named graphs. This flexibility accommodates diverse application requirements for tracking contextual information.

This schema design ensures the storage and retrieval of versioned RDF data while maintaining the flexibility required for diverse application scenarios.

IV. Development

A. Condensed Algebra and SQL Translation

The QuaQue component translates SPARQL queries into SQL by first converting the SPARQL query into a SPARQL algebra expression, and then mapping the SPARQL algebra operators to SQL operations on our condensed model, as shown in Figure 2.

The core of our approach lies in how we handle the validity bitstring. For operations that combine quads, such as joins, we use bitwise operations on the validity bitstrings. For example, a join between two quad patterns corresponds to a bitwise AND operation on their validity bitstrings. This allows us to efficiently determine the versions in which the joined pattern is valid.

The translation process can be summarized as follows (see Figure 2):

- **SPARQL to SPARQL Algebra:** The incoming SPARQL query is parsed into a SPARQL algebra expression tree using Apache Jena.
- **SPARQL Algebra to Condensed SQL:** The SPARQLtoSQLTranslator traverses the algebra tree and, for each operator, generates a corresponding SQL query fragment. The QuadPatternSQLOperator handles the base case of translating a quad pattern into a SQL query on the `versioned_quad` table. Other operators, such as JoinSQLOperator and GroupSQLOperator, combine these fragments using bitwise operations and other SQL constructs.
- **Finalization:** The FinalizeSQLOperator combines the generated SQL fragments into a single, executable SQL query.

B. Sample Dataset

To illustrate the QuaQue approach, we use a simple versioned RDF dataset about users, their friendships, and their preferences. The dataset consists of three versions, each representing a snapshot of the data at a different point in time.

TABLE I. Sample RDF Quads Across Versions

Subject	Predicate	Object	Graph	Versions
:alice	ex:knows	:bob	:g1	1,2,3
:bob	ex:likes	"pizza"	:g1	2,3
:alice	ex:likes	"sushi"	:g1	1,3
:carol	ex:knows	:alice	:g2	3
:bob	ex:knows	:carol	:g2	2,3

Figure 3 provides a graphical representation of the dataset, while Table II lists the metadata associated with the versioned graphs.

In the table II, resources usually prefixed with `:vng` (e.g., `:vng1`, `:vng2`) serve as identifiers for "Versioned Named Graphs", explicitly linking a named graph to a specific version.

For three versions, the bitstring has three bits (e.g., `111` for all versions, `010` for version 2 only). We illustrate the condensed representation of versioned RDF data in Table III.

TABLE II. METADATA OF VERSIONED GRAPHS

Subject	Predicate	Object
:vng1	v:in-version	1
:vng1	v:version-of	:g1
:vng2	v:in-version	2
:vng2	v:version-of	:g1
:vng3	v:in-version	3
:vng3	v:version-of	:g1
:vng4	v:in-version	2
:vng4	v:version-of	:g2
:vng5	v:in-version	3
:vng5	v:version-of	:g2

TABLE III. CONDENSED REPRESENTATION OF THE VERSIONED QUADS (VERSIONED_QUAD)

id_subj.	id_pred.	id_obj.	id_n_graph	validity
1	6	2	10	111
2	7	4	10	011
1	7	5	10	101
3	6	1	20	001
2	6	3	20	011

1) Quad Pattern Translation: The translation of a SPARQL quad pattern into SQL in the condensed model is straightforward. Each quad pattern is mapped to a SQL query over the `versioned_quad` table, with conditions on the `subject`, `predicate`, `object`, and `graph` columns as specified by the quad pattern. The validity bitstring is always projected, as it encodes the presence of the quad across versions. Each variable in the quad pattern is represented as a column in the SQL result, prefixed with `v$` for variables, `ng$` for named graph variables and `bs$` for bitstring variables.

For example, the SPARQL quad pattern:

Query 1: Example SPARQL quad pattern.

```
?s <ex:knows> ?o ?g .
```

is translated to the following SQL:

Query 2: SQL translation of the example SPARQL quad pattern.

```
SELECT (t0.validity) as bs$g,
   t0.id_subject as v$s,
   t0.id_object as v$o,
   t0.id_named_graph as ng$g
FROM versioned_quad t0
WHERE bit_count(t0.validity) <> 0 AND t0.
    id_predicate =
   (Subquery to get id of <ex:knows>)
```

Given the dataset from Table III and the quad pattern of Query 1, the result of the query is shown in Table IV.

Variables `v$s`, `v$o`, and `ng$g` correspond to the subject, object, and named graph IDs, and `bs$g` is the validity bitstring for each result.

2) Join Operation Translation: The translation of a SPARQL join operation into SQL involves combining the SQL fragments generated for each participating quad pattern. The key aspect of this translation is the handling of the validity bitstrings.

Figure 4: Quad pattern translation to SQL

Input: Quad Pattern qp (from current operator)
Output: String sql_query
Function *TranslateQuadPattern(qp)* **is**
  $select \leftarrow$ "$id\_subject, id\_predicate, id\_object$";
  $from \leftarrow$ "";
  $where \leftarrow$ "";
  /* Check if the quad pattern targets the metadata or a versioned graph */
  if $qp.graph =$ "$defaultgraph$" **then**
    $from \leftarrow$ "$metadata$";
  else
    $select \leftarrow$ $select +$ "$, id\_named\_graph, validity$";
    $from \leftarrow$ "$versioned\_quad$";
    $where \leftarrow$ "$bit\_count(validity) <> 0$";
  end
  foreach t *in* $[qp.subject, qp.predicate, qp.object]$ **do**
    $where \leftarrow where + term\_to\_condition(t)$;
  end
  return "$SELECT$" $+ select +$ "$FROM$" $+$ $from +$ "$WHERE$" $+ where$;
end

TABLE IV. RESULT OF THE QUAD PATTERN QUERY

bs$g	v$s	v$o	ng$g
111	1	2	10
001	3	1	20
011	2	3	20

When two quad patterns are joined, their validity bitstrings are combined using a bitwise AND operation. This ensures that the resulting rows only include versions where both patterns are valid.

For example, consider the SPARQL join of two quad patterns on a shared graph name variable:

Query 3: Example SPARQL join of two quad patterns (join on graph name variable).

```
?s <ex:knows> ?o ?g .
?o <ex:likes> ?liked ?g .
```

This join between two quad patterns where the joined variables are in a condensed representation is translated into SQL as follows:

Query 4: SQL translation of the example SPARQL join.

```
SELECT (t0.validity & t1.validity) as bs$g, t0
    .id_subject as v$s, t0.id_named_graph as
    ng$g, t1.id_object as v$liked, t0.
    id_object as v$o
FROM versioned_quad t0, versioned_quad t1
WHERE bit_count(t0.validity & t1.validity) <>
    0 AND
   t0.id_object = t1.id_subject AND
```

Figure 5: Algorithm of the Join translation

Input: Join Operator *join*
Output: String *sql_query*
Function *TranslateJoin(join)* **is**
  $left \leftarrow translate(join.left\_op)$;
  $right \leftarrow translate(join.right\_op)$;
  /* Align representations of joined variables */
  foreach *joined_var in* $left.vars \cap right.vars$ **do**
    $l\_var \leftarrow left.vars.get(joined\_var)$;
    $r\_var \leftarrow right.vars.get(joined\_var)$;
    if $l\_var.repr < r\_var.repr$ **then**
      $right \leftarrow lower(right, r\_var)$;
    end
    if $r\_var.repr < l\_var.repr$ **then**
      $left \leftarrow lower(left, l\_var)$;
    end
  end
  $select \leftarrow get\_joined\_select()$;
  $from \leftarrow left + "," + right$;
  $where \leftarrow get\_joined\_where()$;
  return $"SELECT" + select + "FROM" + from + "WHERE" + where$;
end

```
t0.id_named_graph = t1.id_named_graph AND
t0.id_predicate = (Subquery to get id of <
    ex:knows>) AND
t1.id_predicate = (Subquery to get id of <
    ex:likes>)
```

Given the dataset in Table III, the result of the join Query 3 is shown in Table V.

TABLE V. RESULT OF THE JOIN PATTERN QUERY

bs$g	v$s	ng$g	v$liked	v$o
011	1	10	4	2

Here, `bs$g` is the bitwise AND of the validity bitstrings for the joined quads, indicating the versions in which both patterns are valid.

Consider another example where the join is between a condensed variable and a non-condensed variable.

Query 5: Example SPARQL join of two quad patterns (join between a condensed and a non-condensed variable).

```
?s <ex:knows> ?o ?g .
?g <v:in-version> ?v <ng:Metadata> .
```

In this case, one of the quad patterns includes a variable that is not condensed (i.e., it does not have a validity bitstring associated with it). This scenario requires a different approach to ensure that the join is correctly represented in SQL. In this situation, we need to first flatten the results of the condensed quad pattern to get the capability to join on the non-condensed variable. The associated SQL translation is as follows:

Query 6: SQL translation of the example SPARQL join between a condensed and a non-condensed variable.

```
SELECT left_table.v$s, right_table.v$v,
    left_table.v$g, left_table.v$o
FROM (SELECT flatten_table.v$s, vng.
    id_versioned_named_graph AS v$g,
    flatten_table.v$o FROM (Quad pattern 1)
    flatten_table JOIN versioned_named_graph
    vng ON flatten_table.ng$g = vng.
    id_named_graph AND get_bit(flatten_table.
    bs$g, vng.index_version - 1) = 1)
    left_table
   JOIN (Quad pattern 2) right_table
      ON left_table.v$g = right_table.v$g;
```

Given the dataset from Table III and Table X, the result of the join query 5 is:

TABLE VI. RESULT OF THE JOIN PATTERN QUERY BETWEEN A CONDENSED AND A NON-CONDENSED VARIABLE

v$s	v$o	v$g	v$v
1	6	20	1
1	6	21	2
1	6	22	3
3	1	24	3
2	3	23	2
2	3	24	3

3) Group Operation Translation: Translating a SPARQL group operation into SQL requires aggregating results according to the specified grouping variables. The GroupSQLOperator achieves this by producing SQL with a `GROUP BY` clause for the grouping variables, along with the necessary aggregate functions for the selected variables. The validity bitstring is incorporated to ensure that the aggregation correctly reflects the versioned nature of the data.

For example, consider the SPARQL query that groups by a variable and counts occurrences:

Query 7: Example SPARQL group operation.

```
SELECT ?o (COUNT(?s) AS ?count)
WHERE {
   ?s <ex:knows> ?o ?g .
}
GROUP BY ?o
```

This translation highlights a concept that has been studied in the context of query rewriting with aggregation. Cohen et al. [16] present methods for rewriting queries with aggregation functions using views, which aligns with our approach [27]. In our translation, the aggregation function counts the number of '1's in the validity bitstring, representing the number of versions in which each object occurs. This group operation is translated into SQL as follows:

Query 8: SQL translation of the example SPARQL group operation.

```
SELECT *, agg0 AS v$count
FROM (SELECT v$o, SUM(bit_count(bs$g)) AS agg0
     FROM (
   Quad Pattern
) gp GROUP BY (v$o)) ext
```

Figure 6: Algorithm of the Group translation

```
Input: Group Operator gp
Output: String sql_query
Function TranslateGroup(gp) is
    subquery ← translate(gp.sub_op);
    select ← "";
    from ← "";
    group_by ← "";
    foreach var in gp.grouped_vars do
        if var.repr = "condensed" then
            var.repr ← "id";
            subquery ← lower(subquery, var);
        end
        select ← select + var.name;
    end
    foreach agg in gp.aggregates do
        select ← select + translate_aggregate(agg);
    end
    from ← "(" + subquery + ")gb";
    group_by ← get_group_by(gp.grouped_vars);
    return "SELECT" + select + "FROM" +
      from + "GROUPBY" + group_by;
end
```

Given the dataset from Table III, the result of the group Query 7 is shown in Table VII.

TABLE VII. RESULT OF THE GROUP OPERATION QUERY

v$o	agg0 = v$count
2	3
1	1
3	2

This result demonstrates that the aggregation correctly counts the total occurrences across all versions, leveraging the bitstring representation to efficiently compute version-aware aggregates.

V. BENCHMARKS

A. Benchmark Setup

To evaluate the performance of QuaQue, we conducted a benchmark comparing our system against Jena, high-performance native RDF triple stores.

The benchmark was conducted on a virtual machine hosted on the PAGODA cloud platform provided by LIRIS [42], offering a stable, high-performance, and controlled environment to ensure the accuracy and reliability of results. We leveraged Docker, a containerization platform, to deploy each component of the benchmark in isolated environments. For evaluation, we used dataset and query workloads from the BEAR benchmarks [43], which supply a diverse range of versioned RDF graphs and queries, allowing assessment of system performance across different data sizes. Utilizing the official BEAR queries ensures our evaluation remains standardized and comparable to previous studies. A fixed memory limit was applied throughout to maintain consistency and comparability of results.

BEAR archives datasets with different versioning policies, including Time-Based (TB) and Change-Based (CB) versioning. For this benchmark, we selected the BEAR-B-day dataset, which employs a Time-Based versioning policy. The BEAR benchmarks focus exclusively on triple pattern and join pattern queries, varying across some index—predicate or predicate and object.

The benchmarking environment used the PAGODA virtual machine provider and Docker as the containerization platform. OpenNebula was used as the cloud management platform to orchestrate the virtual machines and is supervised by KVM (Kernel-based Virtual Machine) hypervisor with a host passthrough configuration. The virtual machine ran Ubuntu 24.04 LTS as the operating system with 500GB of allocated disk space. An AMD EPYC 7443 24-Core Processor (48 threads) @ 2.85 GHz CPUs was utilized for the benchmarking tests. This environment had access to a total of 12 virtual CPU cores and 64GB of RAM.

B. Development

For the relational backend of QuaQue, we utilized **PostgreSQL 15**. To support efficient query answering for any given triple pattern, we implemented a comprehensive indexing strategy inspired by the Hexastore approach [44]. We created composite B-Tree indexes on six permutations of the quad components (Graph, Subject, Predicate, Object). This ensures that the query optimizer can utilize an index-only scan for any combination of bound and unbound variables. The specific index definitions are:

- $(id\_named\_graph, id\_subject, id\_predicate, id\_object)$
- $(id\_named\_graph, id\_subject, id\_object, id\_predicate)$
- $(id\_named\_graph, id\_predicate, id\_object, id\_subject)$
- $(id\_named\_graph, id\_predicate, id\_subject, id\_object)$
- $(id\_named\_graph, id\_object, id\_predicate, id\_subject)$
- $(id\_named\_graph, id\_object, id\_subject, id\_predicate)$

Additionally, there is an index on the `digest` column of the `resource_or_literal` table to speed up lookups of RDF terms.

C. Benchmark Results

The results of the storage consumption and query performance evaluations are presented below and are published in more detail in a Zenodo repository [45].

1) Storage Efficiency: Table VIII compares the disk space usage. Native RDF stores, such as Jena TDB2, are highly optimized for storage, utilizing dictionary encoding to map URIs (Uniform Resource Identifiers) and literals to integers, resulting in a compact footprint (694 MB). QuaQue, despite also employing dictionary encoding, exhibits higher storage consumption (4.7 GB) due to its comprehensive indexing strategy on the relational engine. We also include **QuaQue-flat** in our comparison, which serves as a baseline relational implementation where each quad-version pair is stored separately. This is a known trade-off in relational RDF mapping: trading

storage space (via exhaustive indexing) for query flexibility and performance.

TABLE VIII. STORAGE CONSUMPTION COMPARISON.

Dataset	Policy	Tool	Space (MB)
BEAR-B-day	TB	Jena TDB2	**694.39**
BEAR-B-day	TB	QuaQue-flat	6489.91
BEAR-B-day	TB	QuaQue	4707.63

2) Query Execution Time: Table IX reports the execution times for BEAR-B query templates. To ensure accurate and stable measurements, each query was executed 200 times. The first 50 executions were treated as a warm-up phase to mitigate cold-start effects, and the reported results are the average of the final 150 runs. QuaQue demonstrates better performance, performing better than Jena TDB2 in all observed categories (Join, Predicate-Object, and Predicate queries). A Mann-Whitney U-test confirmed the statistical significance of the following results. This suggests that the overhead of the SQL layer is compensated by the efficiency of the PostgreSQL query planner and the availability of covering indexes.

3) Discussion: The experimental results highlight a trade-off between storage efficiency and query performance that warrants critical analysis. As shown in Table VIII, QuaQue's storage footprint is larger than that of Jena TDB2 (4.7 GB vs. 694 MB). This is a direct consequence of our exhaustive indexing strategy, which maintains six B-Tree indexes to cover all possible quad patterns and the index on the digest of the resource values.

However, this investment in storage yields substantial dividends in query execution time.

Table IX shows that QuaQue achieves improvements of approximately 14% for predicate queries, 6% for predicate-object queries, and 14% for join queries compared to Jena TDB2. While these gains are statistically significant, a nearly sevenfold increase in storage for a 10–15% performance improvement represents a trade-off that may not be justified in storage-constrained environments. Figure 7 presents box plots showing the distribution of query execution times. These plots reveal that QuaQue exhibits lower median times and reduced variability compared to Jena TDB2, which may be valuable in latency-sensitive applications where predictability matters.

These results suggest that the primary value of our approach lies not in raw performance gains over highly optimized native RDF stores, but rather in the flexibility of using standard SQL infrastructure and the potential for integration with existing relational ecosystems. The condensed relational model can serve as an effective backend for versioned knowledge graph querying when storage capacity is not a constraint and when interoperability with relational systems is a priority.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we addressed the challenge of efficiently querying versioned Knowledge Graphs. We introduced QuaQue, a system that bridges the gap between Semantic Web standards and Relational Database Management Systems. Our approach relies on a condensed relational model that uses bitstrings to represent the validity of quads across multiple versions, avoiding data redundancy while enabling efficient cross-version querying.

A. Discussions and Future work

1) Benchmark extension: While our current evaluation provides valuable insights, it is limited to basic query patterns. Future work will extend the benchmark to include aggregate queries for a more comprehensive assessment of real-world workloads. Additionally, we plan to compare QuaQue against other versioning policies, such as Change-Based (CB) and Independent Copies (IC), to better understand the trade-offs between storage efficiency and query performance.

2) Extensive implementation: Standard relational algebra lacks support for recursive queries essential for graph analysis, such as path traversal. To address this, we plan to extend our implementation with advanced operators, such as Agrawal's alpha (α) [7]. This extension will enable efficient execution of complex path-finding and recursive queries, significantly enhancing the system's analytical capabilities.

3) Reproducibility: We emphasize that the approach presented in this paper, implemented in the ConVer-G tool, as well as the benchmark used for evaluation, are fully reproducible. The source code, datasets, and experimental scripts are publicly available. To facilitate verification, we provide a containerized environment (Docker) that automates the setup of the database, the loading of datasets, and the execution of the benchmark queries.

In conclusion, QuaQue represents a step towards robust and scalable management of evolving Knowledge Graphs, offering a practical solution for domains requiring complex, concurrent versioning.

ACKNOWLEDGMENTS

This work, *QuaQue: Design and SQL Implementation of Condensed Algebra for Concurrent Versioning of Knowledge Graphs*, was funded by the Université Claude Bernard Lyon 1 as an ATER position, the IADoc@UDL project, and supported by the LIRIS UMR 5205. We would also like to express our sincere gratitude to the BD team and all members of the Virtual City Project [46] for their insightful feedback, constructive discussions, and continuous support throughout the development of this work. Their expertise and collaboration have been instrumental in shaping the direction and quality of this research.

REFERENCES

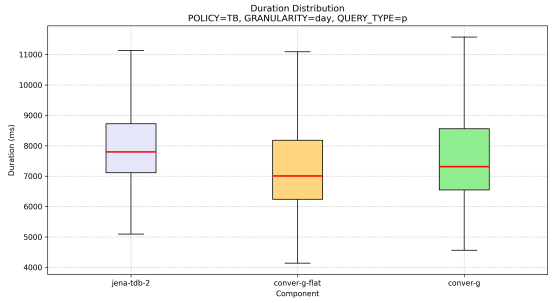
[1] A. e. a. Hogan, "Knowledge graphs", *ACM Computing Surveys*, vol. 54, no. 4, pp. 1–37, 2021.

[2] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust", in *Proceedings of the 14th International Conference on World Wide Web*, 2005, pp. 613–622.

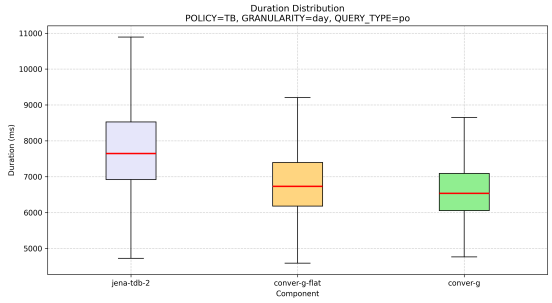
[3] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquière, "Conver-g: Concurrent versioning of knowledge graphs", *arXiv preprint arXiv:2409.04499*, 2024.

TABLE IX. AVERAGE QUERY EXECUTION TIMES (MS).

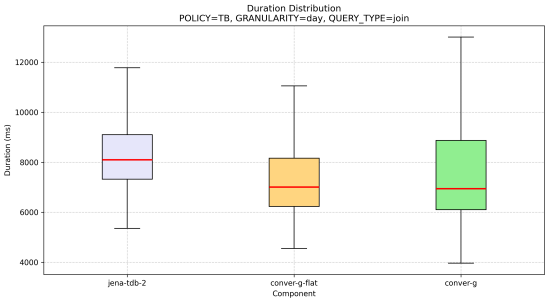
Dataset	Policy	Query type	QuaQue	QuaQue-flat	Jena TDB2
BEAR-B-day	TB	Join	**6936.00**	7006.50	8096.50
BEAR-B-day	TB	P-O	7309.50	**7007.00**	7798.50
BEAR-B-day	TB	P	**6533.50**	6730.50	7644.00



(a) Query Times for predicate index queries



(b) Query Times for predicate-object index queries



(c) Query Times for join queries

Figure 7. Benchmark Results - Query Times

[4] A. e. a. Polleres, "How does knowledge evolve in open knowledge graphs?", *Transactions on Graph Data and Knowledge*, vol. 1, no. 1, pp. 11–1, 2023.

[5] E. F. Codd, "A relational model of data for large shared data banks", *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[6] M. A. Roth, H. F. Korth, and A. Silberschatz, "Extended algebra and calculus for nested relational databases", *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 4, pp. 389–417, 1988.

[7] R. Agrawal, "Alpha: An extension of relational algebra to express a class of recursive queries", *IEEE Transactions on Software Engineering*, vol. 14, no. 7, pp. 879–885, 2002.

[8] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos, "Extending relational algebra and relational calculus with set-valued attributes and aggregate functions", *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 4, pp. 566–592, 1987.

[9] J. D. Ullman, "Implementation of logical query languages for databases", *ACM Transactions on Database Systems (TODS)*, vol. 10, no. 3, pp. 289–321, 1985.

[10] M. Hofer, D. Obraczka, A. Saeedi, H. Köpcke, and E. Rahm, "Construction of knowledge graphs: State and challenges", *arXiv preprint arXiv:2302.11509*, 2023.

[11] X. e. a. Jiang, "On the evolution of knowledge graphs: A survey and perspective", *arXiv preprint arXiv:2310.04835*, 2023.

[12] N. e. a. Abbas, "Knowledge graphs evolution and preservation – a technical report from isws 2019", *arXiv preprint arXiv:2012.11936*, 2020.

[13] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, "A survey on knowledge graphs: Representation, acquisition, and applications", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 494–514, 2021.

[14] A. Gonzalez-Hevia and D. Gayo-Avello, "Leveraging wikidata's edit history in knowledge graph refinement tasks", *arXiv preprint arXiv:2210.15495*, 2022.

[15] H. Zhong, D. Yang, S. Shi, L. Wei, and Y. Wang, "From data to insights: The application and challenges of knowledge graphs in intelligent audit", *Journal of Cloud Computing*, vol. 13, no. 1, p. 114, 2024.

[16] S. Cohen, W. Nutt, and Y. Sagiv, "Rewriting queries with arbitrary aggregation functions using views", *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 2, pp. 672–715, 2006.

[17] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible query processing in starburst", in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989, pp. 377–388.

[18] E. F. Codd, "Relational completeness of data base sublanguages", in *Data Base Systems*, Prentice-Hall, 1972, pp. 65–98.

[19] R. Cyganiak, "A relational algebra for sparql", *Digital Media Systems Laboratory HP Laboratories Bristol, HPL-2005-170*, vol. 35, no. 9, 2005.

[20] D. e. a. Calvanese, "Ontop: Answering sparql queries over relational databases", *Semantic Web*, vol. 8, no. 3, pp. 471–487, 2016.

[21] I. Düntsch, "Relation algebras and their application in temporal and spatial reasoning", *Artificial Intelligence Review*, vol. 23, no. 4, pp. 315–357, 2005.

[22] Z. e. a. Li, "Temporal knowledge graph reasoning based on evolutional representation learning", in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 408–417.

[23] J. F. Allen, "Maintaining knowledge about temporal intervals", *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[24] A. G. Cohn, B. Bennett, J. Gooday, and N. M. Gotts, "Qualitative spatial representation and reasoning with the region connection calculus", *GeoInformatica*, vol. 1, no. 3, pp. 275–316, 1997.

[25] R. Taelman, H. Takeda, M. Vander Sande, and R. Verborgh, "The fundamentals of semantic versioned querying", in *SSWS2018, the 12th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2018, pp. 1–14.

[26] R. Conradi and B. Westfechtel, "Version models for software configuration management", *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, 1998.

[27] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquiere, "Condensed representation of rdf and its application on graph versioning", *arXiv preprint arXiv:2506.21203*, 2025.

[28] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "A fundamental approach to model versioning based on graph modifications: From theory to implementation", *Software & Systems Modeling*, vol. 13, no. 1, pp. 239–272, 2014.

[29] M. Graube, S. Hensel, and L. Urbas, "R43ples: Revisions for triples", in *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)*, 2014.

[30] M. Völkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak, "Semversion: A versioning system for rdf and ontologies", in *Proceedings of the 2nd European Semantic Web Conference (ESWC)*, 2005, pp. 193–202.

[31] M. e. a. Vander Sande, "R&wbase: Git for triples", *LDOW*, vol. 996, 2013.

[32] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquiere, *Conver-g: Concurrent versioning of knowledge graphs*, arXiv:2409.04499 [cs.DB], 2024. arXiv: 2409.04499 [cs.DB].

[33] J. Anderson and A. Bendiken, "Transaction-time queries in dydra", *MEPDaW/LDQ@ESWC*, vol. 1585, pp. 11–19, 2016.

[34] S. Gao, J. Gu, and C. Zaniolo, "Rdf-tx: A fast, user-friendly system for querying the history of rdf knowledge bases", in *Proceedings of the 19th International Conference on Extending Database Technology (EDBT)*, 2016, pp. 269–280.

[35] A. Cerdeira-Pena, A. Fariña, J. D. Fernández, and M. A. Martínez-Prieto, "Self-indexing rdf archives", in *2016 Data Compression Conference (DCC)*, 2016, pp. 526–535. DOI: 10.1109/DCC.2016.40

[36] T. Neumann and G. Weikum, "X-rdf-3x: Fast querying, high update rates, and consistency for rdf databases", *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2, pp. 256–263, Sep. 2010. DOI: 10.14778/1920841.1920877

[37] K. Kulkarni and J.-E. Michels, "Temporal features in sql:2011", *SIGMOD Record*, vol. 41, no. 3, pp. 34–43, Oct. 2012, ISSN: 0163-5808. DOI: 10.1145/2380776.2380786

[38] N. Arndt, P. Naumann, N. Radtke, M. Martin, and E. Marx, "Decentralized collaborative knowledge management using git", *Journal of Web Semantics*, vol. 54, pp. 29–47, 2019, Managing the Evolution and Preservation of the Data Web, ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2018.08.002

[39] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary rdf representation for publication and exchange (hdt)", *Journal of Web Semantics*, vol. 19, pp. 22–41, 2013.

[40] I. Cuevas and A. Hogan, "Versioned queries over rdf archives: All you need is sparql?", in *MEPDaW@ ISWC*, 2020, pp. 43–52.

[41] T. Yang, Y. Wang, L. Sha, J. Engelbrecht, and P. Hong, "Knowledgebra: An algebraic learning framework for knowledge graph", *Machine Learning and Knowledge Extraction*, vol. 4, no. 2, pp. 432–445, 2022.

[42] LIRIS, "PAGODA cloud platform", Accessed: 2025.12.01. [Online]. Available: https://projet.liris.cnrs.fr/pagoda/latest/

[43] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "BEAR – benchmark for rdf archive versioning systems", Accessed: 2025.12.01. [Online]. Available: https://aic.ai.wu.ac.at/qadlod/bear.html

[44] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for semantic web data management", *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[45] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquiere, "Quaque benchmark results", Accessed: 2025.12.01. [Online]. Available: https://zenodo.org/records/17780464

[46] VCity Team, "Virtual city project", Accessed: 2025.12.01. [Online]. Available: https://projet.liris.cnrs.fr/vcity/

APPENDIX

A. QuaQue: A Queryable Versioned Quad Store

1) Sample Dataset: The metadata about versions and graphs is represented in Table X.

TABLE X. REPRESENTATION OF THE METADATA (METADATA)

id_subject	id_predicate	id_object
20	8	1
20	9	10
21	8	2
21	9	10
22	8	3
22	9	10
23	8	2
23	9	11
24	8	3
24	9	11

The following Table XI provides the dictionary mapping for resources and literals.

TABLE XI. DICTIONARY REPRESENTATION OF THE RESOURCES OR LITERALS (RESOURCE_OR_LITERAL)

id_resource_or_literal	name	type
1	:alice	resource
2	:bob	resource
3	:carol	resource
4	"pizza"	literal
5	"sushi"	literal
6	ex:knows	resource
7	ex:likes	resource
8	v:in-version	resource
9	v:version-of	resource
10	:g1	resource
11	:g2	resource
20	:vng1	resource
21	:vng2	resource
22	:vng3	resource
23	:vng4	resource
24	:vng5	resource