Leveraging Query Decomposition for Scalable SPARQL Materialization in Dataspaces

Maarten Vandenbrande[1], Thibeau Vercruyssen, Pieter Bonte[2] and Femke Ongenae[1]

[1]*IDLab, Ghent University - imec, Belgium*
[2]*Departement of Computer Science, KU Leuven Kulak, Belgium*

Abstract

In dataspaces data is often shared and integrated from multiple heterogeneous and distributed sources, querying over these large datasets becomes computationally expensive. This challenge is exacerbated by the fact that SPARQL queries frequently overlap, as they often involve shared subqueries, especially in the context of federated and distributed data sources. The result is that materializing answers to such queries leads to large numbers of redundant materialized views, requiring substantial resources for storage and constant maintenance with limited reuse. This redundancy becomes particularly problematic as materializing views for individual queries can only answer those specific queries, leaving little opportunity for reusing common results from overlapping subqueries. To address this, we propose an algorithm that analyzes a SPARQL query in order to rewrite and decompose it into a collection of subqueries whose materialized views can be concatenated to form the answer to the original query. This way, the materialization of a query results in a number of materialized subquery views that can be re-used more often, thereby reducing the space taken up by materialized views of queries. We evaluate the correctness, resource usage, and execution time of our proposed algorithm on well-known benchmarks, such as the BSBM, LDBC-SNB, and SP²Bench. Our results demonstrate that the proposed implementation can significantly reduce redundancy and improves performance in most cases, making it an effective solution for optimizing query execution in dataspaces, where the need for efficient data retrieval and minimal resource usage is paramount.

Keywords

Query optimization, SPARQL, Query rewriting, Query materialization, Query decomposition

This paper integrates with the W3C Dataspaces Community Group by addressing the generic dataspaces issue https://github.com/w3c-cg/dataspaces/issues/1.

1. Introduction

In dataspaces, organizations and individuals share and integrate data from multiple sources, enabling distributed querying over a federated ecosystem. This brings forth that queries need to be performed over large amounts of data [1, 2, 3, 4]. Because data consumers frequently execute similar SPARQL queries over the same shared datasets, storing the results of these queries as materialized views provides an effective solution to ensure efficient and timely query results [3, 4, 5]. However, when dealing with similar but not identical queries, creating separate materialized views for each query can lead to substantial inefficiencies in computation and storage. Overlapping query results often get recomputed and stored multiple times, increasing redundancy. For instance, in public transport networks, different data consumers have varying needs regarding delay information. A regional transit authority might need to analyze delays across all forms of public transport (buses, trains, and trams) to optimize scheduling and improve efficiency. Meanwhile, a navigation app provider service may only care about train and tram delays, as these impact route recommendations for commuters. Similarly, a local bus operator might only track bus-specific delays for internal performance monitoring. Despite their differences, all these queries share a common subquery for retrieving delay data, leading to redundant computations

The Third International Workshop on Semantics in Dataspaces, co-located with the Extended Semantic Web Conference, June 01, 2025, Portorož, Slovenia

maarten.vandenbrande@ugent.be (M. Vandenbrande); Thibeau.Vercruyssen@UGent.be (T. Vercruyssen); pieter.bonte@kuleuven.be (P. Bonte); femke.ongenae@ugent.be (F. Ongenae)
0009-0003-6470-9123 (M. Vandenbrande); 0000-0002-8931-8343 (P. Bonte); 0000-0003-2529-5477 (F. Ongenae)

and storage when materialized separately. Without an optimized query decomposition approach, each data consumer would effectively duplicate parts of the materialization process.

This article proposes a new approach to mitigate such redundancy through query rewriting and decomposition based on only a single SPARQL query. Specifically, a query is rewritten and decomposed based on union operations. The resulting subqueries can be materialized separately and recombined on demand through simple concatenations of their results. In the public transport example, our approach would create separate subqueries for the delays in each mode of public transport. The data consumers that need the information of all modes of transport would then receive the recombined results of all these subqueries, whilst others get results that leverage the individual subqueries. By reusing a set of shared materialized subquery views instead of creating a large separate view for each query, this approach significantly reduces both storage requirements and maintenance overhead. To achieve our approach, we introduce a set of rewriting rules and algorithms in this article, ensuring that queries are transformed into a normal form suitable for decomposition while preserving their results. Once rewritten, the queries are decomposed into subqueries that can be materialized independently, allowing new queries that share those subqueries to reuse existing results. This leads to a "virtual view" for each query, which points to the relevant subquery views instead of storing yet another result, minimizing duplication in the materialization layer.

Query decomposition and rewriting has been explored in a few areas to increase performance of query execution, either to decrease the computational expensiveness or to decrease the memory used. Decomposition is also used to improve query scalability by performing queries in parallel on multiple nodes [6, 7]. In federated querying, where queries are executed over a number of endpoints, query decomposition has also proven to be useful. Here, query decomposition is used to only query parts of the query that the endpoint has a response for. This decreases the overall query cost [8, 9, 10]. All approaches described above are data-driven, i.e. the decomposition of the query is determined by the organization of the data. In our approach, the decomposition is query-driven in order to promote re-use of results in partially overlapping queries. That being said, the other approaches are able to decompose their queries on all operations and recombine them, whilst our approach only decomposes on unions. Finally, our approach tries to preemptively split the queries into logical subqueries to increase materialized view reuse without knowing what other queries will be asked in the future.

The remainder of this article is structured as follows. Section 2 provides an in-depth description of the methodology, including details on the rewriting rules, the decomposition algorithm, and the mechanics of materializing rewritten queries. Section 3 outlines the experimental setup, where queries sourced or derived from established RDF benchmarks (BSBM, LDBC-SNB, SP^{2}Bench) are used. Section 4 presents empirical results illustrating the benefits of the proposed technique in terms of cache efficiency and overall storage reduction. Section 5 discusses key observations and trade-offs that emerge from these experiments. Finally, Section 6 offers concluding remarks and suggests directions for future research.

2. Methodology

This section introduces the methodology used to develop the rewriting and decomposition algorithms. Decomposing queries based on unions allows for efficient reconstruction of the materialized views, as it is a simple concatenation. However, for this to work, it is essential that no other operations appear after the union. Since queries rarely have a union operation as the top-level operand, they need to be rewritten to elevate the union operations to the top level, ensuring that the query can be properly decomposed. To enable this efficient rewriting, each query is first translated from its SPARQL syntax into a hierarchical tree structure or query plan. Next, the query is rewritten to move unions to the top of this tree. As query execution occurs from bottom (leaf) to top (root), we refer to this process as the *Union Deferring Rewriting System* (UDRS). Finally the query can be decomposed into subqueries, each corresponding to a part of the original query that is now separated by the union operation. The following subsections present the rewriting rules, the rewriting algorithm, and the decomposition algorithm. To clarify the algorithms in these subsections, we continue the example made in the introduction in the field of public transport. Listing 1 provides the SPARQL query for this example, it queries the buses, trains, and trams and filters out those that were not on time.

```
PREFIX ex: <http://example.com/>
SELECT ?route ?type ?status WHERE {
    {
        # Bus BGP
        ?route rdf:type ex:BusRoute .
    }
    UNION
    {
        # Train BGP
        ?route rdf:type ex:TrainRoute .
    }
    UNION
    {
        # Tram BGP
        ?route rdf:type ex:TramRoute .
    }

    # Time BGP
    ?route ex:scheduledDeparture ?scheduledTime .
    ?route ex:actualDeparture ?actualTime .

    FILTER(?scheduledTime != ?actualTime)
}
```

Listing 1: SPARQL query that checks if public transport routes are on time.

2.1. Binary Tree Query Representation

In order to facilitate the process of query rewriting and optimization, it is useful to represent a SPARQL query in an algebraic form:

Definition 1. *The algebraic form of a query is represented as a rooted binary tree, wherein each internal node represents either a unary or binary algebraic operator, with the number of children corresponding to the arity of the operator. The order of these children matters only if the operator is non-commutative. Meaning the Left-Hand Side (LHS) and Right-Hand Side (RHS) operators can't be swapped. Finally, each leaf represents a Basic Graph Pattern (BGP) operand that is composed of one or more triple patterns.*

Remark that the root of a query tree does not have to represent a projection, i.e. the tree may represent only part of a query. This representation encodes algebraic operator precedence within its hierarchy. According to SPARQL's bottom-up semantics, each operator is evaluated only after its operands. This means that a node is only evaluated after all its children. This allows any subtree to be conceptually replaced by a leaf representing its (future) solution, and vice versa. Queries are expressed in their algebraic form, while tree terminology is used to reference the hierarchical relationships within them. Figure 1 illustrates the algebraic tree representation of the query in Listing 1.

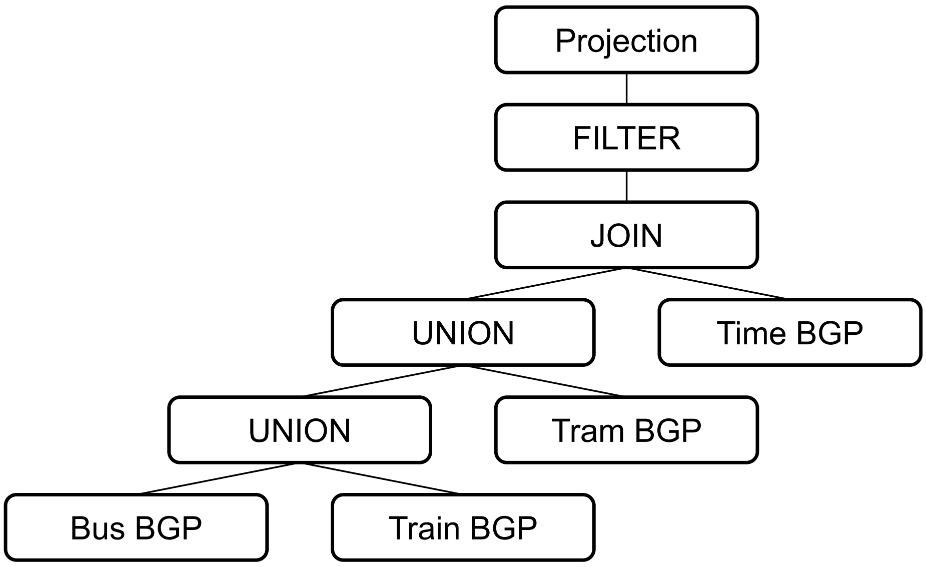

Figure 1: The binary query tree representation of the example query from Listing 1.

2.2. The Rewriting System

The process of rewriting consists of systematically applying all applicable rules from a set of rules, referred to as a rewrite system, to the given input. Applying a rule consists of first identifying an instance of a rule's pattern and then substituting it with an instance of the rule's corresponding replacement pattern. We provide the definition of an *Abstract Rewrite System* (ARS):

Definition 2. *An ARS is a 2-tuple $(X, \rightarrow)$, where X is a set and $\rightarrow$ is a binary relation on X that is a subset of the cartesian product $X \times X$.*

We write $x \rightarrow x'$ $(x, x' \in X)$ to denote that x' is derivable from x by a single rule application, and $x \xrightarrow{*} x'$ to denote derivability by zero or more applications. The normal form of x is x' if $x \xrightarrow{*} x'$ and no rule is still applicable to x'. In other words, there does not exist an $x'' \in X$ such that $(x', x'') \in \rightarrow$. Based on set of query trees representing valid algebraic expressions in the Binary Tree Query Presentation from the previous section, we define an ARS, referred to as the *Union Deferring Rewriting System* (UDRS), to defer the evaluation of union operators whose ancestor operators are only unions. The objective is to maximize the number of union nodes that have only union operator ancestors. A query tree is in normal form when it is no longer possible to decrease the depth of any union node such that it could have only union operator ancestors.
Each rewrite rule that decreases the depth of a union node does so by applying a property based on the arity of the operator its parent represents. Unary parent operators must preserve the union operator, while binary parent operators must exhibit left, right, or both distributive over the union operator. Below, we provide rewrite rules for all algebraic operators that satisfy these properties. After, we also specify the cases where these properties are not valid. The subtrees of the children that do not change when applying the property are represented by their (future) solution Ω.

Definition 3 (Union Deferring Rewriting System). *Let $\Omega_1, \Omega_2,$ and Ω_3 be multisets of solution mappings, and $expr$ be an expression, and PV be a set of query variables.*

__R1__: The filter operator preserves the union operator

$$\sigma_{expr}(\Omega_1 \cup \Omega_2) \to \sigma_{expr}(\Omega_1) \cup \sigma_{expr}(\Omega_2)$$

__R2__: The join operator is distributive over the union operator

$$(\Omega_1 \cup \Omega_2) \bowtie \Omega_3 \to (\Omega_1 \bowtie \Omega_3) \cup (\Omega_2 \bowtie \Omega_3)$$
$$\Omega_1 \bowtie (\Omega_2 \cup \Omega_3) \to (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \bowtie \Omega_3)$$

__R3__: The left join operator is right-distributive over the union operator

$$(\Omega_1 \cup \Omega_2) ⟕_{expr} \Omega_3 \to (\Omega_1 ⟕_{expr} \Omega_3) \cup (\Omega_2 ⟕_{expr} \Omega_3)$$

__R4__: The minus operator is right-distributive over the union operator

$$(\Omega_1 \cup \Omega_2) - \Omega_3 \to (\Omega_1 - \Omega_3) \cup (\Omega_2 - \Omega_3)$$

__R5__: The projection operator preserves the union operator

$$\pi_{PV}(\Omega_1 \cup \Omega_2) \to \pi_{PV}(\Omega_1) \cup \pi_{PV}(\Omega_2)$$

Note that the only combinations of operator and union operands for which no corresponding rule is provided are the minus or left join operator with a union operation as the RHS operand, since neither one of these is left-distributive over the union operator. Also, a union operator with a union operation on the LHS or RHS is not rewritten, as this does not decrease the depth of the union operators. Below, we defined these patterns that can't be rewritten:

Definition 4 (Invalid Rewrite Patterns). *Let $\Omega_1, \Omega_2,$ and Ω_3 be multisets of solution mappings, and $expr$ be an expression.*

*The left join operator is **not** left-distributive over the union operator*

$$\Omega_1 ⟕_{expr} (\Omega_2 \cup \Omega_3)$$

*The minus operator is **not** left-distributive over the union operator*

$$\Omega_1 - (\Omega_2 \cup \Omega_3)$$

*The union operator over the union operator is **not** rewritten*

$$(\Omega_1 \cup \Omega_2) \cup \Omega_3$$
$$\Omega_1 \cup (\Omega_2 \cup \Omega_3)$$

As stated in Definition 1, each internal node of the binary query tree represents only a single algebraic operator. An important drawback of this representation is that a sequence of n consecutive union operators is represented by n union nodes. Consequently, if they are all rewritable (which is logically equivalent to one being rewritable), then the rewrite process consists of at least n rewrite sequences, since each rewrite sequence repeatedly rewrites only one union node. Therefore, we should generalize our representation so that a single internal node represents all consecutive identical associative operators within an algebraic subexpression. So, as an optimization, we can group subsequent operations into one, creating an n-ary query tree. The proofs of the correctness of this optimization are out of scope of this article. Figure 2 shows this optimization applied to the binary query tree in Figure 1. In this query, the two consecutive unions can be grouped together into a ternary union operation.

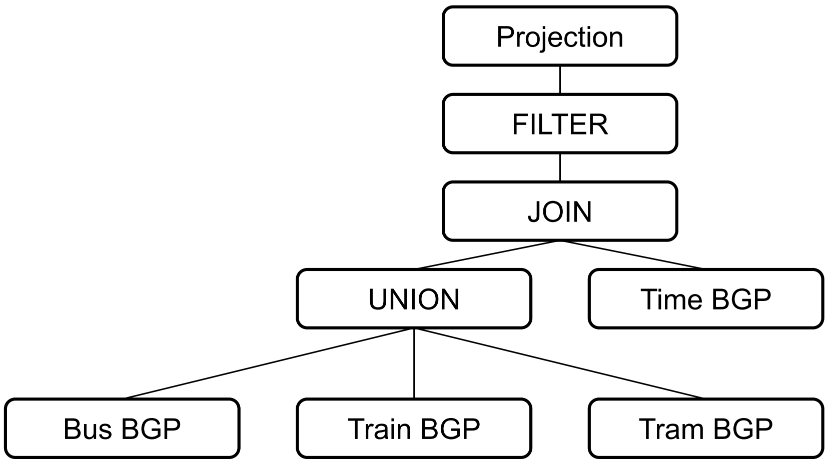

Figure 2: The optimized query tree representation of the example query from Listing 1.

2.3. Verification of the Rewriting System

We need to fulfill the following properties involved in the verification of rewriting systems, as detailed by Dershowitz et al. [11]:

- **Termination:** No infinite derivations are possible. Hence, each object $x \in X$ has at least one normal form.
- **Confluence:** For all objects $s, t, t' \in X$ such that $s \xrightarrow{*} t$ and $s \xrightarrow{*} t'$, there exists an object u such that $t \xrightarrow{*} u$ and $t' \xrightarrow{*} u$. In other words, for any two diverging sequences of rewriting steps, there exists an object u to which both sequences can be rewritten. Hence, every object has at most one normal form.
- **Soundness:** Applying a rule of the system, only rewrites an input $x_1 \in X$ to an output $x_2 \in X$ that is equal to it.

To prove that a rewriting algorithm that uses the UDRS to rewrite an algebraic expression terminates irrespective of the chosen rewrite strategy, we demonstrate that it is terminating.

Property 1 (Termination of the Union Deferring Rewriting System)**.** *The UDRS is terminating.*

Proof. We prove this by contradiction. Assume the existence of an infinite derivation. As shown, every rewrite step of the derivation decreases the depth of one union node by one. Therefore, the existence of an infinite derivation requires that the sum of the depths of all the union nodes is infinite, necessitating an infinite number of nodes within the query tree. However, the number of nodes within the query tree is finite. □

To prove that the normal form of an algebraic expression is independent of the rewrite strategy used in the rewriting process, we prove that the UDRS is confluent.

Property 2 (Confluence of the Union Deferring Rewriting System)**.** *The UDRS is confluent.*

Proof. We use Newman's lemma [12] that states that a terminating ARS is confluent if it is locally confluent. An ARS is locally confluent if for all objects $a, b, c \in X$ such that $a \rightarrow b$ and $a \rightarrow c$, there exists $d \in X$ such that $b \xrightarrow{*} d$ and $c \xrightarrow{*} d$.
Let

- $a \rightarrow b$ be the application of rule r_1, denoted by $\underset{1}{\rightarrow}$, to the subtree s_1 .
- $a \rightarrow c$ be the application of r_2, denoted by $\underset{2}{\rightarrow}$, to the subtree s_2 .

Given the structure of a rewrite rule's pattern, it follows that the maximum overlap between two pattern instances, s_1 and s_2, occurs when s_2 is either a child subtree (operand) of s_1's highest level union node or the non-union child subtree (operand) of s_1's top-level operator node, or vice versa. Furthermore, each rule treats these subtrees as indivisible units. Therefore, applying r_1 will not modify s_2, but might create a copy of it, and vice versa.
a rewritten to b can be rewritten to d as follows:

$$\begin{cases} a \underset{1}{\to} b \overset{2}{\underset{2}{\to}} d & r_1 \text{ applies left- or right-distributivity} \\ & \text{and } s_2 \text{ is the non-union child subtree of } s_1\text{'s top-level operator} \\ a \underset{1}{\to} b \underset{2}{\to} d & \text{otherwise} \end{cases}$$

a rewritten to c can be rewritten to d as follows:

$$\begin{cases} a \underset{2}{\to} c \overset{2}{\underset{1}{\to}} d & r_2 \text{ applies left- or right-distributivity} \\ & \text{and } s_1 \text{ is the non-union child subtree of } s_2\text{'s top-level operator} \\ a \underset{2}{\to} c \underset{1}{\to} d & \text{otherwise} \end{cases}$$

□

Since the UDRS is both terminating and confluence, it follows that each query has exactly one normal form. Hence, exhaustively applying all rewrite rules applicable to a query always yields the same resulting query, regardless of application order.
Finally to prove that applying a rewrite rule of the UDRS—and hence, exhaustively applying all rewrite rules applicable to a query—is sound, we leverage on the proofs from Schmidt et al. [13]. Two query trees (input and output of rewriting) are equal if the algebraic expressions they represent are equal. These expressions are equal if the multisets of solution mappings they produce when evaluated on the same RDF graph are equal. Hence, the UDRS is sound if applying any rewrite rule of it to a query tree results in a query tree that is equal to the original.

2.4. The Query Decomposition and Rewriting Algorithm

Having established that the order of application of applicable rewrite rules used during the rewriting process does not affect the normal form of the query tree, we have the flexibility to freely select the rewrite strategy. The sole requirement is to apply all applicable rules such that the normal form is reached. Consequently, each node must be visited at least once, as every node may be part of a rewrite rule's pattern instance. An effective and simple way to find all rule pattern instances (subtrees) is to locate all union nodes and then examine the parent operator node and determine which side the union is on. This is due to the fact that each rule's pattern instance includes a union operation and, as noted before in Definition 4, only four of parents and child union operator nodes do not constitute rule patterns. As such, we traverse the query tree using Depth First Search (DFS) rather than Breadth First Search (BFS) because after applying a rewrite rule, a new rule can be applied to the union node included in the rule's pattern instance and its new parent, which was previously its grandparent. Off course, as stated previously, except for the cases in which the new parent does not exist, is a union node, or is a minus or left join node that is RHS of its parent. Therefore, maintaining a path from the root enables applying a sequence of rewrites that rewrite the union to the top-level if possible, without revisiting nodes. Hence, adopting DFS is preferred as it inherently maintains this exploration path during traversal. The structure of the rewriting algorithm is shown in Algorithm 1, with the helper function $isRewritable$ defined in Algorithm 2.

```
Input: A query tree (root)
$excludedUnions \leftarrow \{\}$
while $union \leftarrow$ DFS on root for a union in the query tree that is $\notin excludedUnions$ do
    while $isRewritable(PathTo(union))$ do
        $union \leftarrow rewrite(union, parent(union))$
    end
    $excludedUnions \leftarrow excludedUnions \cup \{union\}$
end
```
Algorithm 1: The Union Deferring Rewriting System Rewriting Algorithm

Where $PathTo$ is a function that returns the path from the root to the given node, $isRewritable$ is a function that checks if there exists an applicable rule, and $rewrite$ is a function that applies the applicable rule and returns a reference to the union operator that has been rewritten upwards.

```
Input: A union path (unionPath)
Output: boolean
if $unionPath.parent = \cup$ then // check parent of union
    return false
end
for $i \leftarrow 0$ to $unionPath.length - 1$ do
    if $unionPath[i] \in \{-, ⟕\} \wedge unionPath[i+1]$ is the RHS of $unionPath[i]$ then
        return false
    end
end
return true
```
Algorithm 2: isRewritable(unionPath)

The root of the tree in Figure 2 is the projection. From here, a DFS will go down the tree and encounter the union node on the left of the join. As the path to the union does not include minus or optional and the parent is not a union, it is rewritable. The first rule from Definition 3 we can apply is **R2**, this will rewrite the union above the joins. This results in a union of three joins between the bus, train, or tram BGP's and the time BGP. After, the union can be rewritten above the filter by applying **R1**. Finally, **R5** can be applied to rewrite the union above the projection, the resulting tree after all these rewriting steps is shown in Figure 3.

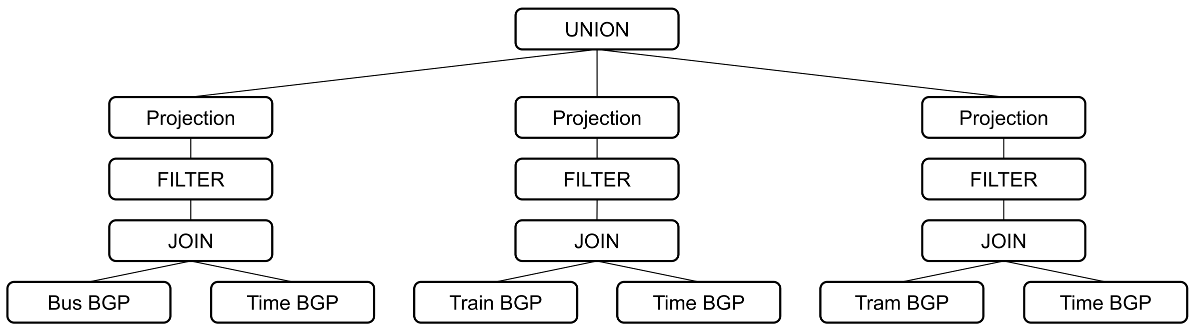

Figure 3: The rewritten version of the query tree representation of the example query from Listing 1.

This query tree now consists of three separate queries that can easily be decomposed into separate queries and individually executed, and the result can then be recombined by simple concatenation. Although uncommon that a select query doesn't have a projection on top, the individual projections would still give a correct query response.

2.5. Query Materialization

The process of materializing a query consists of executing it, storing its result, and continuously maintaining that. We associate the query's query tree representation with this view to identify it. We store the query result and its tree representation together in a state referred to as the materialization state. The current implementation of the algorithm does not support maintaining views, i.e. updating the result as the underlying data changes. However, the functionality to do it is orthogonal to our current implementation and can, hence, be implemented later without altering or affecting any of the existing functionality, with for example incremental query engines like Incremunica [14].

To associate a given query with a materialization state, we first need to determine whether the query (tree) was already materialized previously by determining whether it is equivalent to any materialized query tree. Determining whether two query trees are equivalent is performed by statically determining whether their root nodes are equivalent. Determining the equivalence of two (root) nodes can be done recursively. The base case is whether the set of triple patterns of two BGP leaf nodes are equal. The recursive case consists of determining whether the children of the two nodes are equal, ordered if the identical operation the nodes represent is non-commutative (associativity is not considered as it is beyond the scope of this research), and if any other properties they have are equivalent. These properties are expressions used as conditions in filters or left joins and sets of projection variables, of which only the in-scope ones must be equal.

If the query tree is equal to an already materialized one, we directly return the already materialized answer (cache hit). In case no equal materialized views are found (cache miss), the query needs to be executed. Without query decomposition this would mean executing the complete query using a query engine and storing the result together with the tree representation of the query as a materialization state, such that other queries can re-use the results. This materialization technique is referred to as Full Query Materialization (FQM).

However, by utilizing the query rewriting and decomposition techniques outlined previously in this article, we can achieve a more re-usable and resource efficient algorithm, called Decomposed Query Materialization (DQM). A given query is rewritten and decomposed into a multiset of subqueries, which are executed in parallel since each is independent. The results of these subqueries are stored and materialized through virtual views identified by their query trees. The original query is also materialized via a virtual view that references the materialized views of its subqueries and concatenates them on demand. If the query is not split into at least two subqueries, materializing its single subquery suffices to materialize the entire query, rendering the final step redundant. Remark that a subquery may already be materialized through multiple (sub)subqueries, leading to the query's virtual view referencing another virtual view. However, this can be avoided by incorporating all references contained in that virtual view.

Without query decomposition, the materialized view from the query in Listing 1 would contain all buses and trains that are delayed. Other data consumers that require only the delayed buses cannot use this materialized view. So for the original data consumer, a virtual view is established that combines the result from the two queries. Whilst for other data consumers that require delays for buses and metro's, the query can be decomposed as well where the buses result can be reused from the original data consumer and the metro result can then be calculated in a separate materialized view. Meaning the new data consumer only needs to calculate part of their query, increasing performance.

3. Evaluation Setup

In this section, we outline the evaluation strategy used to compare Full Query Materialization (FQM), i.e. the methodology that is currently used in dataspaces, and Decomposed Query Materialization (DQM), i.e. the algorithm proposed in this article. The code for the implemented query rewriting and materialization algorithm can be found at: https://github.com/KNowledgeOnWebScale/query-decomposition. This repository also contains all the code to execute the evaluations described below.

3.1. Datasets

We source realistic queries and the datasets to execute them on from other RDF benchmarks and execute them against a locally hosted SPARQL endpoint. Specifically, we select relevant benchmarks from the RDF Store Benchmarking W3C Wiki[1]: The BSBM [15]; The LDBC-SNB [16], where test queries are sourced from the Business Intelligence [17] and Interactive [18] workload; and SP^{2}Bench [19]. The remaining ones are considered irrelevant for one or more of the following reasons: their source code is not or no longer public, they provide or use no queries, none of their queries contain at least one union, or they use query-mined queries that would require too much manual attention.

3.2. Evaluation Scenarios

We create four scenarios for each query selected from the datasets. In each scenario the query or one or more derived versions of it are materialized. Each scenario is parameterized by a materialization technique and consists of materialization processes that each have two input parameters: a query and a materialization state.
The first scenario consists of materializing the original query with an empty materialization state, and is referred to as the Full Query (FQ) scenario. It demonstrates a cache miss for both materialization techniques, i.e. FQM and DQM, and thus requires the most work to materialize the query. The resulting materialization state is used as input for the remaining scenarios.
In the second scenario, called Already Materialized Full Query (AMFQ) scenario, the original query is executed again to show a cache hit for both techniques.
Third, the Only One SubQuery (OOSQ) scenario executes each subquery that is derived from rewriting and decomposing the original query. This constitutes a cache miss for the FQM case (as the subquery is not a match with the complete query), but will lead to a cache hit in the DQM case (as it will match on a decomposition).
Finally, the Changed One SubQuery (COSQ) scenario consists of materializing one query variant for each subquery identified through rewriting and decomposition of the original query. The ith variant is derived from the original query such that the ith subquery of its decomposition differs from the original query's decomposition, showing a partial cache hit for DQM and a cache miss for FQM.

3.3. Deriving Test Queries From The Selected Datasets

Each test query must meet the following criteria: the query should be a SELECT query and contain at least one rewritable union node. Additionally, the query's execution must yield a result with more than a handful of solution mappings. As materializing and caching queries with small results is useless, as computing the result is easier than finding the equal query. But the result size should remain below the realistic limit of 10000 solution mappings. This realistic limit was adopted from Virtuoso[2]. For each eligible query in the selected datasets, we first assess whether it meets the criteria stipulated above. If this is the case, we use it directly. If it is not the case, we transform the query such that it meets the criteria, i.e. by changing the query type to SELECT, removing unsupported operators like aggregations, order-by, and limit. The scenario run metrics of queries derived from the same source query are averaged. As such, 5 queries were maintained for LDBC-SNB, 1 query from BSBM and 1 query for SP^{2}Bench. The complete list of used queries and data can be found in this canonical citation: https://doi.org/10.5281/zenodo.15015062.

[1]https://www.w3.org/wiki/RdfStoreBenchmarking
[2]https://virtuoso.openlinksw.com/

4. Results

We measure two metrics for each scenario run: the execution time of each step in the materialization process and the space taken up by (or size of) the materialization state after completing the run. All scenarios are run using Node.js v21.1.0 under WSL on a machine with an Intel® Core™ i5-7400 CPU and 16 GB of DDR4 RAM in dual-channel configuration. Each scenario is run twice for warmup and then ten times to gather metrics. Each displayed metric value represents the average value across these ten runs. Each FQM and DQM scenario run's metrics are reported in Tables 1, 2, and 3, where each single cell is formatted as $^{\text{FQM}}/_{\text{DQM}}$.'-' denotes that the metric was not measured because of the absence of its associated step in that scenario run. In the benchmark queries the average number of children (operands) of a union node, which also represents the average number of subqueries in a decomposition, consists of two numbers because of the LDBC-SNB Q_3 outlier, which has 14 union operands. The overall average is 4.00, while the average without this outlier is 2.33.

Table 1
Comparison of Full-Query and Decomposed Query Materialization in LDBC-SNB

	Step	Scenario			
		FQ	AMFQ	OOSQ	COSQ
Q_1	Size of views (bytes)	1465238/1465270	1465238/1465270	2197857/1465270	2197857/1465294
	Size of query trees (bytes)	1981/5042	1981/5042	3511.5/5042	3963/8555.5
	Total (ms)	126.92/141.87	1.8/2.37	64.99/1.31	71.39/18.31
	Rewrite query tree (ms)	−/7.91	−/–	−/–	−/8.91
Q_2	Size of views (bytes)	44703/44735	44703/44735	67054.5/44735	67054.5/44759
	Size of query trees (bytes)	3338/7117	3338/7117	5227.5/7117	6677/12346.5
	Total (ms)	14.26/26.83	2.27/2.49	8.27/1.32	11.27/17.29
	Rewrite query tree (ms)	−/9.1	−/–	−/–	−/8.18
Q_3	Size of views (bytes)	498118/498342	498118/498342	533697.86/498342	960656.14/498462
	Size of query trees (bytes)	33273/228545	33273/228545	47221/228545	66519/275740
	Total (ms)	734.13/63795.34	16.65/38.37	1230.47/16.48	747.47/2083.47
	Rewrite query tree (ms)	−/766.3	−/–	−/–	−/752.93
Q_4	Size of views (bytes)	2730256/2730288	2730256/2730288	4095384/2730288	4095384/2730312
	Size of query trees (bytes)	5439/15433	5439/15433	10436/15433	10851/25843
	Total (ms)	303.15/425.57	3.88/4.23	193.05/3.15	167.53/82.86
	Rewrite query tree (ms)	−/19.02	−/–	−/–	−/16.25
Q_5	Size of views (bytes)	12531/12563	12531/12563	18796.5/12563	18796.5/12587
	Size of query trees (bytes)	3569/9229	3569/9229	6399/9229	7139/15630
	Total (ms)	12.5/29.34	2.34/2.79	8.42/1.92	11.84/20.25
	Rewrite query tree (ms)	−/9.58	−/–	−/–	−/8.12

Table 2
Comparison of Full-Query and Decomposed Query Materialization in BSBM

	Step	Scenario			
		FQ	AMFQ	OOSQ	COSQ
Q_1	Size of views (bytes)	526027/526091	526027/526091	657533.75/526091	920547.25/526131
	Size of query trees (bytes)	2207/6679	2207/6679	3325/6679	4415/10006
	Total (ms)	42.2/130.44	1.79/1.85	13.71/1.13	32.96/17.16
	Rewrite query tree (ms)	−/9.67	−/–	−/–	−/8.05

Table 3
Comparison of Full-Query and Decomposed Query Materialization in SP²Bench

	Step	Scenario			
		FQ	AMFQ	OOSQ	COSQ
Q_1	Size of views (bytes)	$^{103084}/_{103116}$	$^{103084}/_{103116}$	$^{154626}/_{103116}$	$^{154626}/_{103140}$
	Size of query trees (bytes)	$^{4795}/_{10392}$	$^{4795}/_{10392}$	$^{7593.5}/_{10392}$	$^{9591}/_{17987.5}$
	Total (ms)	$^{26.6}/_{46.6}$	$^{3.2}/_{3.88}$	$^{14.77}/_{1.88}$	$^{19.88}/_{24.02}$
	Rewrite query tree (ms)	$^{-}/_{12.86}$	$^{-}/_{-}$	$^{-}/_{-}$	$^{-}/_{11.25}$

5. Discussion

This section discusses and describes the general observations that can be made from the results table. While network overhead is negligible with our local SPARQL endpoint, it can be significant in practice. This renders the comparison of query rewriting to total materialization time slightly misleading here, as rewriting constitutes a larger portion of the total time than it would in practice.
The total materialization state size overhead in the FQ and AMFQ scenario is 8.32% (without Q3) or 36.79% (with Q3). While in the OOSQ and COSQ scenarios, the DQM state size is 25.46% (without Q3) or -0.25% (with Q3) smaller than the FQM state (the latter being caused by the large number of subquery trees). Hence, we observe that if any subquery's materialized view is used to obtain an answer to a query other than the original query, the size of the materialization state is smaller than if FQM were used, provided that the number of subqueries is not excessive. The OOSQ and COSQ scenario are cache hit and partial cache hit for DQM, while both are cache misses for FQM, forcing it to create new materialized views that duplicate already materialized data. Thus, the size of the FQM materialized views grows larger than that of DQM. Demonstrating that the views created by DQM can be used to (partially) answer queries that the views created by FQM cannot.
Initially materializing a query using DQM is, on average, 193.22% (without Q3) or 8689.93% (with Q3) slower (the latter figure being because of having to materialize 14 subqueries). The execution time taken by DQM compared to FQM, when they both have a cache hit, is, on average, 15.76% (0.39s) (without Q3) or -130.42% (21.72s) (with Q3) longer (the latter again being caused by the large number of subqueries), which is attributed to the larger number of views to check against. While in the OOSQ cache hit scenario, it is 89.43% (without Q3) or 98.66% (with Q3) faster, and in the COSQ partial cache hit scenario, it is 4.59% (without Q3) or -178.73% (with Q3) faster. The latter being a mere 4.59% faster is because, for queries with short execution times, the time taken to rewrite and decompose constitutes 46.93% of the total execution time. The latter figure is due to the execution of the non-materialized subquery taking significantly longer than the full query.
We observe that the time spent rewriting and decomposing a query is significant, constituting, on average,18.63% (without Q3) or 1.28% (with Q3) (FQ scenario) and 42.73% (without Q3) or 38.34% (with Q3) (COSQ scenario) of the total execution time. Moreover, we observe a clear correlation, especially for LDBC-SNB Q_3, between the effort required to rewrite a union node and the quantity and complexity of its ancestors, which are duplicated by the rewriting process. However, rewriting and decomposing can reduce the time taken by the rest of the materialization process on (partial) cache hits, depending on the time needed to answer non-materialized subqueries (done in parallel). The second component of the time overhead of DQM compared to FQM is that materializing all subqueries in parallel may take longer than executing the entire query.
Our analysis demonstrates that while query decomposition and rewriting (DQM) introduce a notable overhead in initial materialization and execution time, they offer significant benefits in reducing storage redundancy and improving cache utilization. The trade-off between query execution time and storage efficiency is particularly evident in scenarios with frequent cache hits, where DQM outperforms full query materialization (FQM) by leveraging reusable subquery results. However, the effectiveness of DQM is influenced by the complexity of query decomposition, particularly in cases with a high number

of unions to rewrite and decompose. Despite these challenges, our results indicate that DQM minimizes redundant materialization, improves reuse of query results, and enhances performance in cache hit scenarios, making it a viable strategy for optimizing query execution in dataspaces.

6. Conclusion

In this work, we aimed to reduce the total size of materialized views used in data spaces by increasing their reusability. We achieved this by mitigating a portion of the overlap between the views of related SPARQL queries using our proposed methodology. This methodology consisted of materializing the solution of each subquery, resulting from first rewriting and then decomposing the query by union operators whose ancestor operators were only other unions and the top-level projection. Subsequently, it involved creating a virtual view for the query itself that referenced those subquery views. Consequently, when queries whose decompositions shared subqueries were materialized, the result was virtual views that referenced the same corresponding subquery views, thereby reducing the total size of the materialized views. We developed, implemented, and integrated the rewriting and the decomposition algorithm into the process of query materialization, deriving a new materialization technique termed DQM. We evaluated this new technique against (direct) Full Query Materialization (FQM) and showed two of the situations in which the views created by Decomposed Query Materialization (DQM) can be used to (partially) answer queries that the views created by FQM cannot. Notably, forcing FQM to create new materialized views that duplicate already materialized data. Although our implementation struggled with queries with a lot of unions, in most situations the DQM returned query answers faster than FQM.
Future work should focus on improving the performance of the query rewriting process, particularly for queries with extensive union operations. As demonstrated in our results, the current implementation does not perform optimally when faced with a high number of unions, leading to increased rewriting overhead. Optimizing the rewriting algorithm to handle deep or complex union structures more efficiently could significantly enhance the overall materialization process.
While our approach focuses on decomposing queries based on union operations, future research could explore decomposition strategies based on other SPARQL algebraic operations.
Finally, extending the implementation to support a broader range of SPARQL algebra operators would improve its applicability. Currently, our method does not fully support operations such as aggregations, ORDER-BY, LIMIT, ect. Incorporating these features would allow for a wider variety of queries to benefit from decomposition-based materialization. Handling aggregations efficiently, for instance, would require the development of specialized decomposition techniques that allow incremental aggregation computation over shared subqueries.

Acknowledgments

This work was partly funded by the SolidLab Vlaanderen project (Flemish Government, EWI and RRF project VV023/10) and the FWO Project FRACTION (Nr. G086822N).

Declaration on Generative AI

During the writing of this paper, the author(s) used DeepL and GPT-4o in order to: Grammar, translation and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication’s content.

References

[1] M. Franklin, A. Halevy, D. Maier, From databases to dataspaces: a new abstraction for information management, SIGMOD 34 (2005) 27–33. URL: https://dl.acm.org/doi/10.1145/1107499.1107502. doi:10.1145/1107499.1107502.

[2] A. Halevy, M. Franklin, D. Maier, Principles of dataspace systems, in: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, 2006, pp. 1–9. URL: https://dl.acm.org/doi/10.1145/1142351.1142352. doi:10.1145/1142351.1142352.

[3] S. Song, L. Chen, M. Yuan, Materialization and decomposition of dataspaces for efficient search, IEEE 23 (2011) 1872–1887. URL: https://ieeexplore.ieee.org/abstract/document/5611525. doi:10.1109/TKDE.2010.213, conference Name: IEEE Transactions on Knowledge and Data Engineering.

[4] M. Jarke, C. Quix, Federated data integration in data spaces., Designing Data Spaces 181 (2022). URL: https://library.oapen.org/bitstream/handle/20.500.12657/57901/978-3-030-93975-5.pdf#page=190.

[5] M. Vandenbrande, Aggregators to realize scalable querying across decentralized data sources, in: ISWC2023, the International Semantic Web Conference, 2023. URL: https://biblio.ugent.be/publication/01HFXXKQ51BYBEMG4Z01EK4CH1/file/01HFXXN3ZQNVKX0M4J5MFHP2RH.

[6] J. Huang, D. J. Abadi, K. Ren, Scalable SPARQL querying of large RDF graphs, VLDB 4 (2011) 1123–1134. URL: https://doi.org/10.14778/3402707.3402747. doi:10.14778/3402707.3402747.

[7] B. Wu, Y. Zhou, P. Yuan, L. Liu, H. Jin, Scalable SPARQL querying using path partitioning, in: 2015 IEEE 31st International Conference on Data Engineering, 2015, pp. 795–806. URL: https://ieeexplore.ieee.org/abstract/document/7113334. doi:10.1109/ICDE.2015.7113334, ISSN: 2375-026X.

[8] K. M. Endris, P. D. Rohde, M.-E. Vidal, S. Auer, Ontario: Federated query processing against a semantic data lake, in: S. Hartmann, J. Küng, S. Chakravarthy, G. Anderst-Kotsis, A. M. Tjoa, I. Khalil (Eds.), Database and Expert Systems Applications, Springer International Publishing, Cham, 2019, pp. 379–395.

[9] S. Nural, P. Koksal, F. Ozcan, A. Dogac, Query decomposition and processing in multidatabase systems, in: OODBMS Symposium of the European Joint Conference on Engineering Systems Design and Analysis, Montpellier, Citeseer, 1996.

[10] D. Suciu, Query decomposition and view maintenance for query languages for unstructured data, in: VLDB, volume 96, Citeseer, 1996, pp. 3–6.

[11] N. Dershowitz, Computing with rewrite systems, Information and Control 65 (1985) 122–157.

[12] M. H. A. Newman, On theories with a combinatorial definition of" equivalence", Annals of mathematics 43 (1942) 223–243.

[13] M. e. a. Schmidt, Foundations of sparql query optimization, in: Proceedings of the 13th International Conference on Database Theory, ICDT '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 4–33. doi:10.1145/1804669.1804675.

[14] V. Maarten, T. Ruben, B. Pieter, O. Femke, Incremunica: Web-based incremental view maintenance for sparql, in: The Semantic Web: 22th International Conference, ESWC 2025, June 1–5, 2025, Proceedings 22, Springer, 2025.

[15] C. e. a. Bizer, The berlin sparql benchmark, International Journal on Semantic Web and Information Systems (IJSWIS) 5 (2009) 1–24.

[16] R. A. et al., The LDBC Social Network Benchmark, CoRR abs/2001.02299 (2020).

[17] G. S. et al., An early look at the LDBC Social Network Benchmark's Business Intelligence workload, in: GRADES-NDA at SIGMOD/PODS, ACM, 2018, pp. 9:1–9:11.

[18] O. E. et al., The LDBC Social Network Benchmark: Interactive workload, in: SIGMOD, 2015, pp. 619–630.

[19] M. e. a. Schmidt, Spˆ 2bench: a sparql performance benchmark, in: 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 222–233.