

A Hybrid Vector-Symbolic Data Model for Multimodal Knowledge Graphs

Florian Ruosch 

FPR Consulting, Zurich, Switzerland

Luca Rossetto 

Dublin City University, Dublin, Ireland

Abstract

In this article, we address the architectural and representational “schism” between symbolic knowledge graphs and sub-symbolic multimodal data. By formalizing the *MediaGraph* Data Model, we demonstrate that high-dimensional vectors and multimedia segments can be integrated as first-class citizens within a unified graph structure. This shift from treating vectors as opaque literals to queryable nodes enables “early-binding” query execution, allowing the SPARQL engine to jointly optimize over both structural relationships and perceptual similarity. Our implementation, *MeGraS*, provides a scalable engine that mitigates the long-standing $N + 1$ query problem through batched execution. The experimental evaluation on both real-world lifel-

ogging data and a synthetic dataset of 10^7 triples confirms that our hybrid approach maintains high throughput and retrieval efficiency, even as the complexity of multimodal queries increases. Furthermore, the integration of the *Unified Multimedia Segmentation* Model ensures that media granularity is preserved, making complex spatiotemporal and content-aware querying possible within a single query framework. Ultimately, *MeGraS* provides a foundational infrastructure for the next generation of multimodal intelligent systems. By unifying the symbolic and sub-symbolic domains, we provide a pathway toward more expressive and efficient retrieval strategies that can keep pace with the explosion of heterogeneous digital media.

2012 ACM Subject Classification Information systems → Users and interactive retrieval; Information systems → Graph-based database models; Information systems → Multimedia and multimodal retrieval

Keywords and phrases Multimodal Knowledge Graphs, Graph Store, Multimodal Media Segmentation

Digital Object Identifier 10.4230/TGDK.4.3.3

Category Research

Supplementary Material *Software (System)*: <https://github.com/MediaGraphOrg/MeGraS> [15]

archived at `swh:1:dir:b0d522ad9603f43d5135aa393c24d74d7ea8a3d7`

Dataset: <https://github.com/MediaGraphOrg/MeGraS-SYNTH> [16]

archived at `swh:1:dir:98b200e81f058bd739629f0f2dca77cb7ea523e6`

Funding This work was partially funded by the Swiss National Science Foundation through Project “MediaGraph” (Grant Number 202125).

Received 2026-03-04 **Accepted** 2026-08-26 **Published** 2026-09-30

Generative AI Declaration The authors used generative AI (Anthropic Claude: Opus 4.8, Opus 5, Fable 5, and Fable 5.1) in the preparation of this article: to assist in drafting the manuscript text from the authors’ own outlines, notes, and results; for language and style improvement; to assist with the revision in response to the reviews, including checking the manuscript against the released source code; and to assist in writing software code, including the code from which Figures 1 and 2 were generated. All references were compiled manually, and all experimental data stem from the authors’ own measurements. All AI-assisted text, code, and figures were manually reviewed, verified, and edited by the authors, who take full responsibility for the entire content of the article.



© Florian Ruosch and Luca Rossetto;
licensed under Creative Commons License CC-BY 4.0

Transactions on Graph Data and Knowledge, Vol. 4, Issue 3, Article No. 3, pp. 3:1–3:28



Transactions on Graph Data and Knowledge

TGDK Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The digital landscape is increasingly defined by the explosion of multimodal data: a heterogeneous mix of symbolic metadata, high-dimensional vectors, and raw media assets like images and video. While knowledge graphs have long served as the gold standard [1] for representing structured, symbolic relationships via the Resource Description Framework, they have historically struggled to integrate the sub-symbolic intelligence generated by modern AI [8]. Currently, a profound schism exists between symbolic AI, which excels at graph-based reasoning, and sub-symbolic AI, which produces the feature vectors and embeddings used in multimedia retrieval [12].

Traditional approaches to bridging this gap typically treat vectors as opaque literals or external attachments to a graph [11]. This separation forces an “early-binding” or “late-fusion” dilemma for system architectures. They must either perform vector similarity searches in a dedicated vector database and join the results with a symbolic store, or store vectors as binary blobs in a triple store, rendering them invisible to the SPARQL query optimizer. This fragmented architecture leads to significant performance bottlenecks, most notably the $N + 1$ query problem, where a join between symbolic patterns and vector similarity results requires iterative, sequential database calls.

In this article, we propose a shift from treating multimodal data as an extension of symbolic graphs to their unification in the *MediaGraph* Data Model [18]. Our model formalizes a graph structure where Multimedia Document Nodes ($\mathcal{M}$) and Vector Nodes ($\mathcal{V}$) coexist as first-class citizens alongside standard URIs ($\mathcal{U}$), blank nodes ($\mathcal{B}$), and literals ($\mathcal{L}$). This approach allows for “early-binding” query execution, where vector similarity is treated not as an external filter, but as a core relational operation integrated directly into the SPARQL algebra.

To instantiate this model, we leverage *MeGraS* [14], our open-source multimodal knowledge graph store and query engine. *MeGraS* is designed to leverage modern hardware architecture, specifically the parallel processing power of multi-core CPUs and GPU-accelerated embedding computation, achieving the throughput required for real-time multimodal querying.

In summary, we make the following contributions with this article:

- **The formalization of the Unified *MediaGraph* Data Model:** We define a mathematical framework that expands the RDF model to include $\mathcal{M}$ and $\mathcal{V}$ nodes, enabling a more expressive representation of multimodal entities.
- **The *MeGraS* Architecture:** We detail the implementation of a hybrid vector-symbolic engine that utilizes batched execution strategies to address the $N + 1$ query problem in multimodal SPARQL.¹
- **A comprehensive experimental evaluation:** We demonstrate the scalability and efficiency of our approach using a real-world lifelogging dataset and a high-scale synthetic dataset of 10^7 triples.²

Relation to Our Prior Work. This article consolidates and substantially extends a line of work within the *MediaGraph* project. The *MeGraS* system itself was first introduced in a four-page overview [14], which outlined its features but contained neither a formal data model nor a quantitative evaluation. A companion article [18] developed the *content-aware* perspective of the *MediaGraph* Data Model, focusing on multimedia document nodes, their segmentation following the *Unified Multimedia Segmentation* Model [19], and application scenarios; it does not formalize vectors as graph nodes, define a hybrid query algebra, or report any performance measurements.

¹ <https://github.com/MediaGraphOrg/MeGraS>

² <https://github.com/MediaGraphOrg/MeGraS-SYNTH>

Finally, the LifeGraph 5 application paper [17] deployed an earlier *MeGraS* configuration at the Lifelog Search Challenge and explicitly identified the tuple-at-a-time execution of the default Apache Jena pipeline – and the resulting $N + 1$ query problem – as the system’s central unresolved performance bottleneck. The present article provides what these prior publications lack: (i) the formalization of the hybrid vector-symbolic data model, in which Vector Nodes, hybrid triples, and the k -NN operation are defined as elements of a unified algebra (Sections 3 and 4); (ii) the *BatchingQueryEngine*, a set-oriented execution strategy that directly resolves the bottleneck diagnosed in [17]; and (iii) the first systematic experimental evaluation of the approach, comprising the new *MeGraS-SYNTH* benchmark, pushdown and scalability analyses, ingestion throughput, and the dynamic-versus-materialized trade-off (Section 6).

2 Related Work

The evolution of multimodal knowledge graphs has been marked by a transition from using vectors as purely analytical byproducts to integrating them as fundamental structural components of the graph. Early intersections of graph theory and machine learning primarily focused on knowledge graph embedding [20] techniques, such as DistMult [20] and TransE [2] and its successors, resulting in knowledge graphs in vectors. These models map existing symbolic entities and relations into a low-dimensional vector space to facilitate tasks like link prediction. In this paradigm, the vector space serves as a latent representation of the graph, but the vectors themselves remain external to the queryable data model. Conversely, modern multimodal tasks require a vectors-in-knowledge-graphs approach, where high-dimensional feature embeddings are stored directly as nodes. This enables a unified representation in which mathematical coordinates in a d -dimensional manifold coexist with symbolic URIs, allowing queries that combine structural pattern matching with semantic similarity within a single logical framework, as in the one we propose.

However, a fundamental architectural split exists in how graph databases manage vector data. Most industry-standard property graphs and multi-model databases treat vectors as “properties,” and store embeddings as opaque binary literals or arrays attached to a symbolic node [11]. This model creates a significant barrier for query optimizers: because the vector is treated as a literal attribute, it is often invisible to the relational algebra, necessitating expensive late-binding operations that require the engine to fetch iteratively and parse literals for similarity calculations. We propose a shift toward the vector-as-nodes formalization, elevating vectors within the graph’s set of vertices. By treating vectors as first-class nodes, we enable early-binding query execution and treat vector similarity as a native relational join rather than a post-processing filter.

In the absence of native hybrid engines, architectural decoupling has often been employed to bridge the gap between symbolic and vector data. One approach [10] adopts a “thinking, fast and slow” metaphor and decomposes queries into subqueries optimized for either a vector view or a knowledge graph view. While this dual-view system enables robust inference by combining complementary strengths, it relies on a federated query processing engine to manage the interaction between these distinct representations. In contrast to this decoupled architecture, our approach achieves deeper integration by treating vector similarity as a first-class relational operation. By unifying these domains into a single query engine, we eliminate the synchronization overhead inherent to federated approaches and enable the symbolic context to guide vector-based retrieval more effectively.

Finally, while previous efforts to extend the Semantic Web for multimedia provided a foundation for spatiotemporal querying, they were primarily designed for a metadata-centric paradigm that predates the requirements of modern neural retrieval. SPARQL-MM [6] focuses primarily on the management of media URIs and basic contextual descriptors. However, it lacks a formal

mechanism for treating high-dimensional vectors as queryable entities within the graph’s native algebra. Furthermore, traditional multimodal knowledge graphs often treat media as external references, denying the engine direct access to the underlying content for real-time feature extraction or segmentation. We address the lack of granularity by integrating the *Unified Multimedia Segmentation* [19] model, which ensures that media segments are not merely pointers but are URI-addressable nodes dynamically linked to their visual and semantic feature vectors. By addressing these representational gaps, we provide the expressiveness required for advanced content analysis that traditional knowledge graph extensions for multimodality were not designed to support.

3 Formal Data Model

To overcome the limitations of traditional multimodal knowledge graphs, which treat multimedia data as opaque literals or external references, we introduced the *MediaGraph* Data Model [18]. It extends the standard Resource Description Format (RDF) definition to include multimedia documents and high-dimensional vectors as intrinsic node types. We denote the set of all possible entities and relations in the graph by defining disjoint sets that partition our node-coordinate system.

Let $\mathcal{U}$ be the infinite set of all Uniform Resource Identifiers (URIs). Elements $u \in \mathcal{U}$ provide a globally unique naming mechanism for resources, properties, and classes. In the vector-symbolic graph, $\mathcal{U}$ serves as the primary anchor for the symbolic layer, enabling interoperability with the linked data ecosystem. Let $\mathcal{B}$ be the set of blank nodes, representing existential variables or anonymous resources whose identity is local to the graph. A node $b \in \mathcal{B}$ is used to represent resources for which a global URI has not been assigned, often acting as a bridge for complex metadata structures or intermediate grouping nodes.

The union of these sets, $\mathcal{N}_{sym} = \mathcal{U} \cup \mathcal{B}$, constitutes the symbolic domain of the graph. Traditional RDF stores further include a set of Literals $\mathcal{L}$; however, in our *MediaGraph* model, to enable content-aware querying, we extend this symbolic foundation with Multimedia Document Nodes $\mathcal{M}$, their special case of Segment Nodes $\mathcal{S}$ based on the *Unified Multimedia Segmentation* (UMS) [19] model, and Vector Nodes $\mathcal{V}$.

► **Definition 3.1** (Multimedia Document Nodes). *Let $\mathcal{D}$ be the set of all binary data blobs. A Multimedia Document Node $m \in \mathcal{M}$ is a specialized URI-addressable node that acts as a pointer to a specific element $\beta \in \mathcal{D}$ with an associated MIME type τ . We define a Multimedia Document Node as the triple, where u is a URI:*

$$m = \langle u, \tau, \beta \rangle \in \mathcal{U} \times \text{MIME} \times \mathcal{D} \quad (1)$$

Unlike standard RDF literals, Multimedia Document Nodes are addressable bitstreams that can be passed to feature extraction or segmentation functions at query time.

► **Definition 3.2** (Segment Nodes). *Following the UMS model, a Segment Node $s \in \mathcal{S}$ represents a spatiotemporal portion of a parent Multimedia Document Node $m \in \mathcal{M}$. It is formally defined by a transformation σ :*

$$s = \sigma(m, \Delta) \quad (2)$$

where Δ is a segmentation definition comprising filters (e.g., temporal intervals, spatial regions) and reductions (e.g., channel selection).

While $\mathcal{S} \subset \mathcal{M}$ ensures segments are URI-addressable, their relationship to m is determined by the geometric and temporal constraints in Δ .

► **Definition 3.3** (Vector Nodes). *Let $\mathcal{V}$ be the domain of high-dimensional feature vectors. A Vector Node $v \in \mathcal{V}$ is defined as a point in a d -dimensional Euclidean space:*

$$v = [x_1, x_2, \dots, x_d] \in \mathbb{R}^d \quad (3)$$

where $d \in \mathbb{N}^+$ denotes the dimensionality.

These are stored in optimized formats to avoid the overhead of parsing lexical RDF literals.

By unifying these disparate data types, we can define the comprehensive domain of all possible vertices in our hybrid graph.

► **Definition 3.4** (Node Domain). *The set of all nodes $\mathcal{N}$ in the vector-symbolic graph is defined as the union of symbolic, perceptual, and mathematical domains:*

$$\mathcal{N} = \mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{M} \cup \mathcal{V} \quad (4)$$

where $\mathcal{U}$ are URIs, $\mathcal{B}$ are Blank Nodes, $\mathcal{L}$ are Literals, $\mathcal{M}$ are Multimedia Document Node, and $\mathcal{V}$ are Vector Nodes.

This expanded node domain allows for the construction of triples that bridge high-level semantic concepts with low-level perceptual features.

► **Definition 3.5** (Hybrid Triples). *A Hybrid Triple t_h is a statement that bridges the symbolic and sub-symbolic domains. It is defined as a triple where the object belongs to the vector domain:*

$$t_h \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{M}) \times \mathcal{U} \times \mathcal{V} \quad (5)$$

Hybrid triples allow the graph to directly associate a media document m with its mathematical representation v via a semantic predicate p . Note that hybrid triples deliberately exclude vector subjects: statements whose subject is a vector node – for instance, a materialized similarity link between two vectors, or a dimensionality declaration attached to a vector – are permitted by the general statement model introduced below but are not hybrid in this sense, as a hybrid triple specifically denotes the bridge from the symbolic and perceptual domains into the vector domain.

To support statement-level metadata, like the confidence of a feature extractor or the temporal validity of a segment, we further organize these statements into a **QuadSet**.

► **Definition 3.6** (QuadSet Statement). *A QuadSet Statement $q \in \mathcal{G}$ is a tuple:*

$$q = (s, p, o, i) \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{M} \cup \mathcal{V}) \times \mathcal{U} \times \mathcal{N} \times \mathcal{I} \quad (6)$$

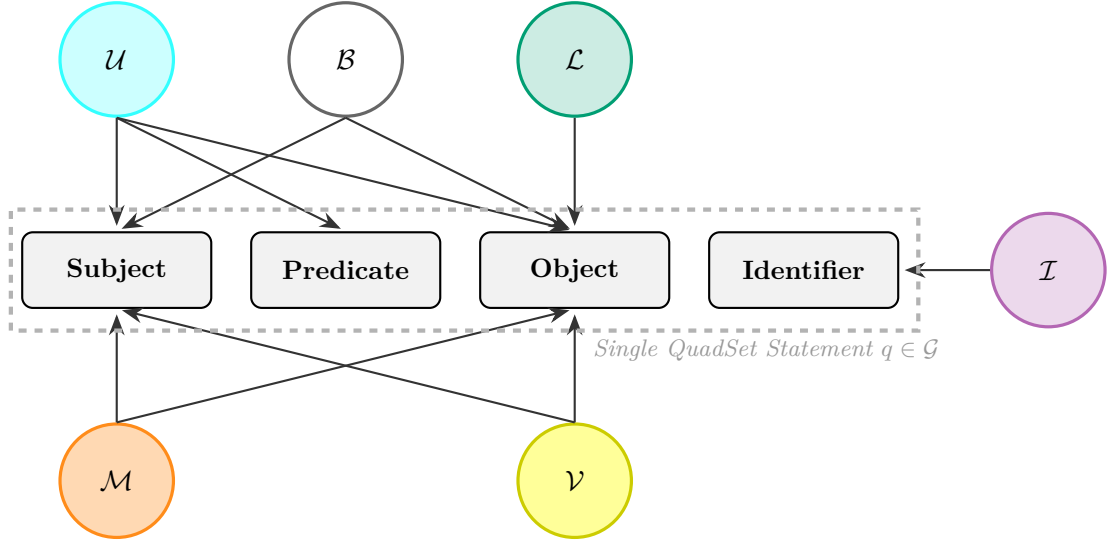
where $i \in \mathcal{I}$ is a unique identifier for the statement, enabling complex reification and referencing.

As illustrated in Figure 1, this structure allows the symbolic domain ($\mathcal{U}, \mathcal{B}, \mathcal{L}$) to provide the structural backbone of the graph, while the perceptual ($\mathcal{M}$) and mathematical ($\mathcal{V}$) nodes are integrated as first-class participants in the subject and object positions.

Hence, we define the formal structure of the *MediaGraph* itself and the fundamental retrieval operation used for multimodal search.

► **Definition 3.7** (MediaGraph). *A MediaGraph $\mathcal{G}$ is a set of QuadSet statements that form a directed labeled multigraph where nodes can represent symbolic, perceptual, or mathematical entities:*

$$\mathcal{G} \subseteq (\mathcal{U} \cup \mathcal{B} \cup \mathcal{M} \cup \mathcal{V}) \times \mathcal{U} \times \mathcal{N} \times \mathcal{I} \quad (7)$$



■ **Figure 1** Visualization of the disjoint set overlap within the **QuadSet** structure. The symbolic domain $(\mathcal{U}, \mathcal{B}, \mathcal{L})$ provides the structural and literal backbone, while $\mathcal{M}$ and $\mathcal{V}$ are integrated as first-class participants.

This structure ensures that every relationship, whether symbolic or sub-symbolic, is uniquely identifiable and queryable through a unified algebra.

Finally, to perform similarity search over the vector nodes, we define a distance function δ that quantifies the dissimilarity between two vectors in the same manifold.

► **Definition 3.8** (Distance Function). *The Distance Function δ maps a pair of vectors to a non-negative real value:*

$$\delta : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}_0^+ \quad (8)$$

Common implementations include the Euclidean distance (L_2) or Cosine distance, depending on the normalization of the embedding space and the requirements of the underlying feature extractor.

By establishing the **QuadSet** as the container for both symbolic metadata and vector nodes, we can define retrieval as a selection process over this unified space. Specifically, integrating the distance function δ into the graph’s topology enables queries that go beyond exact URI matching.

► **Definition 3.9** (Nearest-Neighbor Query). *A Nearest-Neighbor Query Q_{kNN} over the graph $\mathcal{G}$ finds a set of k nodes that are most similar to a query vector v_q under distance function δ :*

$$Q_{kNN}(v_q, k, \mathcal{G}) = \{v \in \mathcal{V} \mid \exists (s, p, v, i) \in \mathcal{G} \text{ and } |\{v' \in \mathcal{V} \mid \delta(v', v_q) < \delta(v, v_q)\}| < k\} \quad (9)$$

By elevating vectors to first-class nodes within the **QuadSet** structure, this operation can be interleaved with symbolic triple patterns. This capability enables “early-binding” query optimization, where similarity search acts as a native relational join rather than an external post-processing filter.

► **Definition 3.10** (Implicit Relations). *An Implicit Relation is a triple $t_i = \langle s, p, o \rangle$ such that $\mathcal{G} \cup \mathcal{R} \models t_i$, where $\mathcal{R}$ represents a set of logical entailment rules. These relations are defined by the graph’s structural logic and are inherently virtual. They are never materialized, as their validity depends on base assertions in $\mathcal{G}$.*

► **Definition 3.11** (Derived Relations). *A Derived Relation is a triple $t_d = \langle s, p, f_p(s) \rangle$, where f_p is a transformation function associated with predicate p . The function f_p can represent symbolic data extraction or sub-symbolic feature extraction. Unlike implicit relations, t_d is materializable or may exist in the graph *ex-ante*. If it does not exist yet upon the first execution of $f_p(s)$, the resulting triple t_d is persisted such that $\mathcal{G}_{new} = \mathcal{G} \cup \{t_d\}$. Subsequent access to t_d incurs no computational cost beyond standard retrieval.*

The implicit and derived relations were introduced and described in [17]. Their inclusion extends the *MediaGraph* Model from a static repository into a dynamic knowledge environment. This distinction is critical for the *MeGraS* architecture: while implicit relations maintain logical consistency without storage overhead, derived relations allow the system to adapt. By materializing derived triples after their first execution, *MeGraS* effectively shifts the incurring “inference tax” from a recurring computational burden to a one-time ingestion cost. This optimization significantly improves long-term query performance and ensures that complex multimodal operations remain scalable.

We emphasize the scope of these constructs: the rule set $\mathcal{R}$ in Definition 3.10 consists of computable structural rules (e.g., k -nearest-neighbor associations, temporal ordering, or segment containment), not ontology axioms. Implicit and derived relations are therefore retrieval mechanisms that compute or look up triples at query time; they do not perform ontological inference or logical entailment in the sense of RDFS/OWL reasoning engines. Layering such reasoners on top of the *MediaGraph* Model is an orthogonal extension that we do not claim in this work.

4 Query Algebra and Execution

To effectively manage the interplay between structured graph patterns and multimedia processing, we introduce an execution framework centered on algebraic integration and batched retrieval. The ensuing sections detail the formal extension of the query algebra, the specialized batching engine, and the heuristic optimizations that enable scalable multimodal retrieval.

4.1 Multimodal Query Algebra and Integration

Since interactions with knowledge graphs heavily rely on structured query languages [5], we extend the standard SPARQL semantics by treating multimedia operations, such as spatiotemporal segmentation, visual feature extraction, and high-dimensional vector similarity searches, as first-class relational operators rather than external function calls. As defined in the core architecture, this enables complex sub-symbolic operations such as `nearestNeighbor` search for high-dimensional vector similarity and `textFilter` for full-text search. By defining these operations at the algebraic level, they can be included in the query graph as joinable nodes rather than post-processing filters.

The engine executes hybrid patterns as a functional composition of symbolic and sub-symbolic constraints. Let σ_{sym} represent symbolic constraints, which act as a filter on the global graph $\mathcal{G}$. The evaluation of a multimodal retrieval pattern is then defined as:

$$\Omega_{out} = \text{kNN}(\sigma_{sym}(\mathcal{G}), \vec{v}_{query}, k) \quad (10)$$

In this formulation, the engine prioritizes evaluating $\sigma_{sym}(\mathcal{G})$ to reduce the search space before accessing the computationally intensive vector index. A key mechanism in this algebra is the construction of a “whitelist” of valid identifiers derived from symbolic filters, which is subsequently pushed into the vector retrieval operator. This ensures that the sub-symbolic similarity search is performed only on the subset of the graph that already satisfies the query’s structural and metadata constraints.

4.2 Batched Execution Strategy

The primary challenge in multimodal graph execution is the $N + 1$ query problem, which occurs when a symbolic join results in a series of sequential calls. As a concrete example, consider the hybrid pattern in Listing 8: a symbolic filter first binds N subjects (e.g., $N = 1,000$); a tuple-at-a-time pipeline then issues one initial call to resolve the filter, followed by one k -NN lookup *per bound subject*; i.e., 1,001 sequential round-trips to the storage layer, each incurring its own connection, parsing, and planning overhead. To mitigate this, the proposed engine implements a set-oriented execution model that replaces the traditional “tuple-at-a-time” pipeline used in standard graph stores. Instead of passing individual bindings through the execution plan, the engine materializes intermediate result sets at critical join points. This materialization phase collects all active bindings for a given triple pattern into a single list, enabling the engine to transform multiple sequential lookups into a single bulk retrieval operation. By interacting with the underlying storage layer through a batched interface, the engine can resolve thousands of potential connections in a single interaction. This approach effectively amortizes the overhead of database round-trips and enables the storage backend to utilize optimized internal join strategies across the entire batch.

4.3 Heuristic Pattern Reordering and Selectivity Scoring

A primary innovation in the execution strategy is the heuristic optimization for basic graph patterns, as described in Algorithm 1, which deviates from the default execution strategy. We utilize a greedy reordering algorithm that scores patterns based on their immediate binding state. This scoring system is based on the structural properties of knowledge graphs, where certain components of a triple are typically more restrictive than others. Following the logic in Algorithm 1, patterns containing text filters are assigned the highest possible weight. This prioritization stems from the fact that text filters, such as keyword searches, are highly selective and can instantly reduce a candidate set by a large margin. On the other hand, constants in the subject position receive a moderate weight, as they represent specific nodes but may have many outgoing edges. Finally, constant predicates receive the lowest weight because they represent the schema level and are often the least selective elements in a graph. As the execution progresses, the engine dynamically updates these scores. Variables bound in earlier steps are treated as “pseudo-constants”, receiving scores that ensure the engine prioritizes “anchoring” the query around known data points before expanding into broader graph traversals. By greedily selecting the pattern with the highest cumulative score at each step, the engine minimizes the memory footprint of the materialized bindings.

The specific weight magnitudes in Algorithm 1 are not individually calibrated quantities but encode a dominance hierarchy. They are chosen such that (i) a text filter (+100) strictly outweighs any combination of the remaining features, which sum to at most +35, guaranteeing that text-filtered patterns are always scheduled first, and (ii) among the remaining patterns, constants are strictly preferred over bound variables in the same position, and positions are ranked by their typical selectivity in knowledge graphs (object > subject > predicate), reflecting the observation that constant predicates operate at the schema level and are usually the least selective triple component [5]. Any weight assignment that preserves these dominance relations yields the same relative ordering for the pattern combinations occurring in our workloads, so the engine’s behavior is insensitive to the exact magnitudes. What the heuristic cannot capture are data-dependent cardinalities; this is precisely the gap that the cost-based optimizer outlined in Section 8 would close.

■ **Algorithm 1** Greedy basic graph pattern reordering heuristics employed.

Require: A set of triple patterns $P = \{p_1, \dots, p_n\}$, and a set of initially bound variables V_{bound}
Ensure: An ordered sequence of patterns S

```

1:  $S \leftarrow []$  ▷ Initialize empty execution sequence
2: while  $P$  is not empty do
3:    $bestScore \leftarrow -1$ 
4:    $bestPattern \leftarrow \text{null}$ 
5:   for each  $p \in P$  do ▷ Assign weights based on constants or already bound variables
6:      $score \leftarrow 0$ 
7:     if  $p$  has associated text filter then  $score \leftarrow score + 100$ 
8:     end if
9:     if  $p.object$  is constant then  $score \leftarrow score + 20$ 
10:    else if  $p.object \in V_{bound}$  then  $score \leftarrow score + 15$ 
11:    end if
12:    if  $p.subject$  is constant then  $score \leftarrow score + 10$ 
13:    else if  $p.subject \in V_{bound}$  then  $score \leftarrow score + 8$ 
14:    end if
15:    if  $p.predicate$  is constant then  $score \leftarrow score + 5$ 
16:    else if  $p.predicate \in V_{bound}$  then  $score \leftarrow score + 3$ 
17:    end if
18:    if  $score > bestScore$  then
19:       $bestScore \leftarrow score$ ;  $bestPattern \leftarrow p$ 
20:    end if
21:  end for
22:  Append  $bestPattern$  to  $S$ ; Remove  $bestPattern$  from  $P$ 
23:   $V_{bound} \leftarrow V_{bound} \cup \text{variables in } bestPattern$  ▷ Update bound variables
24: end while
25: return  $S$ 

```

4.4 Postgres Backend and Physical Execution

The physical execution of these batched queries relies on the robust PostgreSQL-based persistence store. This backend is optimized explicitly for high-volume data retrieval. A critical component in this layer is the adaptive query construction logic, which monitors the size of incoming batches to determine the most efficient SQL translation. For smaller batches below a threshold of 2,000, the store uses a chain of **OR** clauses or **IN** predicates for rapid lookups. However, for larger batches, the system switches to a **VALUES** clause strategy. It constructs a temporary internal table in the database session containing the batch of IDs, then performs a native SQL **JOIN** between the main quads table and this temporary table. This approach leverages the database's ability to use hash joins, which is significantly faster than parsing a massive SQL statement with thousands of individual conditions. Additionally, an abstraction layer provides an in-memory cache for URI and literal mappings, minimizing the need for expensive joins against dictionary tables. This multi-layered approach allows the system to maintain interactive query speeds even when performing complex multimodal joins over millions of edges.

5 Vector Integration Architecture

We leverage *MeGraS* [14] to operationalize and empirically demonstrate our proposed architectural principles and optimization strategies for managing heterogeneous multimodal graph data.¹ *MeGraS* achieves storage-agnosticity through a layered architecture that disconnects RDF logic from physical storage, which can then reside in a tab-separated values file, a PostgreSQL database, or CottontailDB [3]. This decoupling is essential for seamless integration of high-dimensional vectors, enabling the system to treat embeddings as first-class citizens within a unified data model without being tied to a specific database backend. By bridging the gap between structured symbolic relations and sub-symbolic feature representations, the architecture enables the query engine to apply graph-traversal logic to vector nodes. This section details how this layered approach facilitates the representation, persistence, and retrieval of high-dimensional vectors.

5.1 Vector Data Representation

At the core of *MeGraS* is the *MediaGraph* Data Model [18]: a unified type system that extends traditional literal representations to include high-dimensional vectors. Vectors are encapsulated within a specialized class hierarchy (for the numerical precisions `float`, `double`, and `long`) that inherits from the base `QuadValue` type. This type system mirrors the formal model: with Definition 3.6 admitting $\mathcal{V}$ in the subject domain, vectors are first-class statement participants in both the subject and the object position, while predicates remain restricted to URIs; the only position where the implementation’s type system is technically more permissive than the model. In practice, persisted statements host vectors predominantly in the object position (e.g., an embedding attached to a multimedia document); the subject position is exercised primarily in query patterns, for instance, when a k -NN search is anchored on a supplied query vector.

To accommodate the diverse output formats of current machine learning models, the architecture preserves the original precision of the source models, enabling *MeGraS* to avoid quantization errors during ingestion and storage. Furthermore, the system maintains internal metadata on the dimensionality and numerical type of each vector.

5.2 Dynamic Persistence Strategy

For the implementation presented in this work, the persistence layer of *MeGraS* uses the PostgreSQL backend, augmented with the `pgvector`³ extension, to facilitate native high-dimensional operations. While this foundation provides the necessary primitives for vector arithmetic and indexing, its application within a multimodal context necessitates an architectural design capable of managing structural variance. A significant challenge in multimodal knowledge graphs is the heterogeneity of vector lengths, as different extraction functions produce embeddings of varying dimensions within the same graph.

To address this, *MeGraS* implements a dynamic table strategy that partitions vectors by dimensionality and numerical type rather than storing them in a monolithic, sparse structure. As new vector types are encountered during data creation or ingestion, the system automatically generates specialized tables optimized for the specific dimension and precision of the incoming data. This design also enables the use of targeted indices tailored to the specific characteristics of each vector space, ensuring the database engine can maintain optimal execution plans. Specifically, *MeGraS* manages Hierarchical Navigable Small World (HNSW) indices [9] (with construction parameters $m = 16$ and `ef_construction = 64`) using cosine distance on these dynamic tables

³ <https://github.com/pgvector/pgvector>

to facilitate efficient approximate nearest-neighbor searches across diverse feature spaces. By decoupling the primary quad storage from the high-dimensional payload storage, the system maintains the performance of symbolic lookups while enabling low-latency similarity retrieval for multimedia content.

5.3 Persistence-Level Retrieval

The architectural cohesion between symbolic and sub-symbolic data is maintained by exposing vector-specific operations directly through the storage abstraction layer. By implementing specialized operators within the persistence layer, *MeGraS* abstracts the underlying complexity of high-dimensional similarity metrics into a unified interface. This design ensures that the physical proximity of vector data to the indexing structures is leveraged, allowing the system to delegate computationally intensive similarity calculations to the optimized PostgreSQL engine.

Such a strategy is essential for maintaining data locality, as it avoids the substantial latency overhead associated with transferring large embedding vectors to the application layer for manual processing. Furthermore, the persistence architecture is designed to support high-performance hybrid lookups via a low-level filtering interface. This approach enables the storage backend to apply symbolic constraints via SQL joins on the quad table before ordering by vector distance, thereby leveraging both relational indices and HNSW structures. By performing this pruning within the database’s internal execution context, *MeGraS* minimizes the I/O demand on the HNSW structures and ensures that sub-symbolic retrieval remains performant even in highly constrained query scenarios. This integration effectively transforms the persistence layer from a passive data repository into an active component of the hybrid retrieval process.

6 Experiments

In this section, we evaluate the custom query engine of *MeGraS* through a multi-stage experimental framework. We begin by validating system feasibility on real-world lifelogging data, followed by a rigorous performance analysis using synthetic benchmarks to isolate engine-level optimizations, and finally by exploring the overhead of advanced features.

6.1 Setup

To ensure the reproducibility of our timing results, we developed a dedicated benchmarking suite in Kotlin, which interacts with the *MeGraS* instance via its REST-based SPARQL endpoint for query evaluation and directly with the persistence store for ingestion performance. The suite, including all benchmark entry points, query definitions, and the database configuration used for the experiments, is available in the `test` directory of the *MeGraS* repository.¹ All reported results were obtained with release `v0.1` (commit `8c74a91fe9682ea22f7f1e2c5634b4d5cdb0bdea`) of *MeGraS* and release `1.0` (commit `ad173ba2f100aa0f1761a80ef3dc777d814c6f15`) of the *MeGraS-SYNTH* generator.² The execution flow is designed to capture both the engine’s immediate responsiveness and its sustained performance. For the queries, the runner first performs a “cold start” execution. In this context, “cold start” denotes the latency of the very first execution of a specific query within a session. This measurement captures the initial overhead of parsing the query, establishing the HTTP connection, and, most crucially, the time required for the underlying database to fetch uncached index segments and data pages from disk into memory.

After the cold start, the runner performs two more warm-up iterations to stabilize the environment, followed by ten timed warm iterations. Furthermore, the infrastructure (*MeGraS* and its backend) is restarted between runs. The metrics reported are derived solely from these warm runs (except for the cold start) and represent the system’s performance once the relevant graph fragments reside in the database’s cache.

6.1.1 Baselines

The primary baseline for our performance evaluation is an industry standard for RDF and SPARQL processing: Apache Jena.⁴ Standard Jena employs the pipeline model, a sequential pattern-by-pattern execution strategy. When interfacing with external persistence layers, this often leads to the $N + 1$ query problem, where a join between two patterns results in a separate database call for each intermediate result.

To ensure a fair and rigorous comparison that accounts for modern optimization techniques, we implemented a custom `BatchingQueryEngine` within the Jena framework.¹ Instead of traditional nested-loop joins, our batching baseline materializes bindings from the input iterator into bulk sets. These are then dispatched as unified, batched queries to the underlying persistence-layer storage. Note that the `BatchingQueryEngine` is not an independent third-party baseline but the engine contributed in this work, implemented within the Jena framework so that both configurations share the identical SPARQL front end, data model, and persistence store. The comparison is therefore structured as a controlled ablation of the execution strategy: any performance difference between the two configurations is attributable solely to set-oriented batching and operator pushdown, and not to differences in storage layout, vector indexing, or query parsing.

6.1.2 Datasets

MeGraS-SYNTH. To evaluate the scalability and retrieval performance of the system, we generated a synthetic dataset derived from the COCO dataset [7]. The generation process followed a two-stage pipeline: first, we established a base set by selecting a reproducible sample of 1,000 images from the validation 2017 collection, which were all ingested to *MeGraS*. Subsequently, for each image, a CLIP embedding [13] was computed via the *MeGraS* API and stored as a vector node within the graph. In the second stage, we utilized an inflation script to scale the dataset to a target magnitude of 10^7 triples by augmenting each subject with a high volume of noise properties and specifically calibrated selectivity predicates (0.1%, 1%, 10%, 50%). To ensure multidimensional evaluation, the inflator derived vector nodes of varying dimensionality from each base CLIP embedding: truncation for 256 dimensions, the unmodified embedding for 512, and padding with seeded Gaussian noise for 768 and 1,024 dimensions. This synthetic dataset enables controlled benchmarking of index construction times and query latency across different vector-symbolic densities. It is designed for control over selectivity and dimensionality rather than distributional realism; the LSC '25 collection serves as the realistic complement (see Section 8). The code to generate **MeGraS-SYNTH** is available in the online code repository.²

LSC '25. To evaluate *MeGraS* in a realistic retrieval setting, we use the multimodal knowledge graph *LifeGraph 5* [17], which participated in the 8th annual Lifelog Search Challenge (LSC) in 2025 [4].⁵ Table 1 shows an overview of the key numbers of *LifeGraph 5*. At the core of the knowledge graph are the more than 700,000 images, which are described by over 26 million triples. The graph density is relatively high at 17.3 triples per subject, encompassing diverse predicates ranging from symbolic semantic tags and OCR text to vector-based visual embeddings computed with CLIP. For our experiments, the persistence store utilizes a 17 GB PostgreSQL dump of this collection. However, the LSC dataset is not public and can only be obtained by requesting it from the challenge organizers.

⁴ <https://jena.apache.org>

⁵ <https://github.com/MediaGraphOrg/LifeGraph5>

■ **Table 1** Structural statistics of the multimodal knowledge graph *LifeGraph 5* [17] built from the LSC’25 dataset [4] utilized in the real-world feasibility evaluation.

Metric	Value
Total Triples	26,749,381
Multimedia Nodes (Images)	722,086
Vector Nodes	724,736
Distinct Subjects	1,545,949
Distinct Predicates	26
Distinct Objects	4,042,770
Avg. Triples per Subject (Density)	17.30
On-disk Storage Size	17 GB

6.1.3 Hardware and Software Stack

All experiments were conducted on a workstation equipped with an Intel Core i7-12700K processor (12 cores/20 threads, up to 5.0 GHz), 64 GB of DDR4-3200 RAM, and Windows 11 64-bit (Build 26200). The high-dimensional feature embeddings were computed using an NVIDIA GeForce RTX 3080 GPU with 10 GB of dedicated VRAM. The software architecture followed a hybrid deployment model wherein *MeGraS* ran directly in an OpenJDK 19 JVM managed via IntelliJ IDEA, while the persistence store was hosted in a Docker container running PostgreSQL 17.5 with the `pgvector` extension.⁶

To balance application performance and data throughput, the system utilized a tiered storage configuration: the Windows 11 OS, IntelliJ IDEA, and the Docker engine were hosted on a high-speed NVMe SSD, while the WSL2 distribution was located on a mechanical HDD. Crucially, the PostgreSQL data directory was physically bind-mounted back to the NVMe SSD. This setup ensures that the performance-critical persistence layer, specifically high-dimensional vector similarity searches and index-heavy graph lookups, benefits from SSD-level random access speeds and IOPS, effectively isolating the database’s data path from the mechanical latency of the drive hosting the WSL subsystem. We note that this tiered storage configuration is specific to our workstation; absolute latency figures, cold-start times in particular, are therefore not directly transferable to other deployments, whereas the relative comparisons (speedup factors and scaling trends) constitute the portable results of this evaluation.

6.2 System Validation: Real-World Feasibility

We first evaluate *MeGraS* under real-world conditions using the dataset and tasks from the LSC 2025 [4]. This validation serves as a “sanity check” to evaluate the `BatchingQueryEngine` under heterogeneous data conditions. Specifically, we examine its capacity to execute hybrid join operations across noisy OCR output, non-uniform spatiotemporal anchors, and skewed property distributions.

The LSC requires systems to perform interactive retrieval across a large-scale, multimodal lifelogging dataset (images, locations, biometrics), mirroring the challenges of real-world search scenarios. Using a custom-built frontend, we interacted with the knowledge graph store solely via its SPARQL endpoint.

⁶ <https://hub.docker.com/r/timescale/timescaledb-ha>

3:14 A Hybrid Vector-Symbolic Data Model for Multimodal Knowledge Graphs

The tasks in the LSC fall into three categories: Known-Item Search (KIS), where a single specific instance must be found based on an increasingly specific description; ADHOC search, where as many relevant examples as possible must be retrieved, which are assessed by human judges; and visual Question Answering (QA), which requires interactive exploration to answer specific questions about the timeline. We demonstrate the feasibility of *MeGraS* using representative tasks from each type as posed during the LSC '25.⁷ The corresponding SPARQL queries are drawn from the set of queries issued by the user during the live competition, reflecting the unoptimized, iterative nature of real-world interactive retrieval. Table 2 shows the key performance metrics for all four presented SPARQL queries.

The first query example is taken from a KIS task, which reads as follows:

Eating off the floor; I was not used to that. We were sitting on the floor. There were other people eating Asian food with me. I was using chopsticks. I am sure I was in Asia.

At the time of the submission of the query below, only the underlined part of the item description had been available, leading to an unselective criterion through pure-symbolic pattern matching on a single semantic tag.

■ Listing 1 SPARQL query for the KIS task LSC2519.

```
PREFIX lsc: <http://lsc.dcu.ie/schema#>
PREFIX tag: <http://lsc.dcu.ie/tag#>

SELECT DISTINCT ?img ?id ?day
WHERE {
  ?img lsc:id ?id ;
    lsc:tag tag:food ;
    lsc:day ?day .
}
ORDER BY ?id
```

For the KIS task, the engine retrieved 8,322 results with a mean latency of 347.80 ms. This execution involves the bulk retrieval of literal properties for a large set of subjects, where the `BatchingQueryEngine` processes these attributes as a set operation rather than via serial graph traversals.

Next, the QA task presents a complex scenario, necessitating multi-stage retrieval:

I remember being in a taxi with a yellow sign taped to the back of a car headrest that said something like “smile you are on camera”. Where did I go to?

The user first filters for images associated with the location “Car” whose OCR text contains the substring “smile”:

■ Listing 2 SPARQL query for the first part of the QA task LSC2509 (LSC2509_1).

```
PREFIX lsc: <http://lsc.dcu.ie/schema#>
PREFIX tag: <http://lsc.dcu.ie/tag#>
```

⁷ <https://github.com/lucaro/LSC-Archive/tree/main/2025>

```

SELECT DISTINCT ?img ?day ?id
WHERE {
  ?img lsc:id ?id ;
    lsc:day ?day ;
    lsc:location_name "Car" ;
    lsc:ocr ?ocr .
  FILTER(CONTAINS(LCASE(STR(?ocr)), LCASE("smile")))
}
ORDER BY ?id

```

This query bridges symbolic metadata with text-search filters, resulting in a mean latency of 175.00 ms. After identifying the specific taxi segment, the user explores the temporal context to find the destination:

■ **Listing 3** SPARQL query for the second part of the QA task LSC2509 (LSC2509_2).

```

PREFIX lsc: <http://lsc.dcu.ie/schema#>

SELECT DISTINCT ?img ?id ?day
WHERE {
  <http://localhost:8080/EiCqaIJDpr...> lsc:ordinal ?ordinal ;
    lsc:day ?day .
  ?img lsc:id ?id ;
    lsc:day ?day ;
    lsc:ordinal ?ord .
  BIND (100 AS ?n)
  FILTER ((?ord >= (?ordinal - ?n)) && (?ord <= (?ordinal + ?n)))
}
ORDER BY ?id

```

The temporal neighborhood traversal was performed within the specified 100-ordinal range, yielding a mean latency of 74.70 ms.

Finally, for the ADHOC task, the following description was given:

Find examples of when I was paying for something with a currency that is not a EURO.

Since we need to find many, potentially visually similar images, we leverage the system's vector capabilities to identify the 100-nearest neighbors in the CLIP-embedding space of an image known to meet the above criterion, using an implicit relation (see Definition 3.10).

■ **Listing 4** SPARQL query for the ADHOC task LSC2511.

```

PREFIX lsc: <http://lsc.dcu.ie/schema#>
PREFIX implicit: <http://megras.org/implicit/>

SELECT DISTINCT ?img ?id ?day
WHERE {
  <http://localhost:8080/EiCc8JkYsN...> implicit:clip100nn ?img .
  ?img lsc:id ?id ;
    lsc:day ?day .
}

```

■ **Table 2** Performance metrics for representative queries on the LSC '25 dataset [4]. All timing values (Mean, Cold Start, Min, Max, Median, and Std. Dev.) are reported in milliseconds (ms).

Query	Mean	Cold Start	Min	Max	Median	Std. Dev.	Result Count
LSC2519	347.80	929	326	394	342.50	22.05	8322
LSC2509_1	175.00	306	105	286	125.50	73.13	4
LSC2509_2	74.70	145	57	97	70.00	12.93	201
LSC2511	2020.20	2830	1956	2189	2002.00	69.26	100

This is the most computationally demanding query, involving a 512-dimensional k -NN join across more than 720,000 vector nodes. The query recorded a mean latency of 2020.20 ms with a standard deviation of 69.26 ms. In this configuration, the `BatchingQueryEngine` binds the vector similarity join as an early constraint in the execution plan.

Timing and Cold Start Analysis. The performance metrics reported in Table 2 highlight a significant difference between cold-start latencies and subsequent execution times. The cold start captures the initial overhead required for the system to parse the query, establish the database connection, build the execution plan, and, most critically, load the necessary data and index segments from disk into the database’s internal memory buffers.

As seen in the results, the cold start penalty is the highest in the vector-based ADHOC query (Listing 4), reaching 2830 ms, a 40% increase over the mean. For the symbolic KIS query (Listing 1), the cold start time of 929 ms is nearly triple the mean latency. Following these initial runs, latencies for subsequent executions dropped toward the reported median and minimum values.

6.3 Performance Analysis

After the real-world feasibility validation benchmarks, we use `MeGraS-SYNTH` to isolate the impact of our engine-level optimizations. First, we consider three aspects: the cost of hybridity, the pushdown effect, and scalability (graph size and vector dimensionality). Then, we also investigate data ingestion throughput and the advanced feature of dynamic knowledge generation.

6.3.1 The Cost of Hybridity

To evaluate the baseline performance overhead introduced by augmenting symbolic patterns with perceptual similarity operators, we conducted an experiment using the `MeGraS-Synth` dataset at a 10^7 configuration. To ensure a rigorous comparison across the three query types (symbolic-only, vector-only, and hybrid), the result set size was held constant at 10 per test case. This required calibrating the symbolic filter selectivity per query type to match the 10 results of the vector-only 10-NN retrieval: the symbolic-only query uses the 1% predicate (`synth:sel101`), binding ten subjects with one property value each, while the hybrid query uses the 0.1% predicate (`synth:sel1001`), binding a single subject for which the ten nearest neighbors are retrieved. The result set is intentionally kept small and identical across the three query types so that the measurement isolates engine-level coordination overhead rather than result-serialization and transfer costs, which grow with output size for all query types alike. In the following listings, predicates in the `synth:` namespace stem from the `MeGraS-SYNTH` inflation script: `synth:selXXX` denote boolean selectivity predicates that hold for a calibrated fraction of all subjects (e.g., `synth:sel1001` for 0.1% and `synth:sel101` for 1%), while `synth:prop_N` denote generic literal-valued noise properties attached to every subject. The symbolic-only query, which serves as a baseline for standard lookups, was defined as:

■ **Listing 5** Symbolic-only SPARQL query.

```
PREFIX synth: <http://megras.org/synth#>

SELECT ?s ?val
WHERE {
  ?s synth:sel01 "true" .
  ?s synth:prop_0 ?val .
}
```

The vector-only query represents the cost of sub-symbolic similarity retrieval in isolation:

■ **Listing 6** Vector-only SPARQL query.

```
SELECT ?o
WHERE {
  <http://localhost:8080/EiD5nIES6Y...> <http://megras.org/implicit/10nn/vec512> ?o .
}
```

Finally, the hybrid query evaluates the coordination efficiency of the `BatchingQueryEngine` when resolving joint vector-symbolic execution. Its second triple pattern establishes a complex relation that the `BatchingQueryEngine` resolves as a combination of implicit and derived relations (see Definitions 3.10 and 3.11). Specifically, the predicate `http://megras.org/implicit/10nn/vec512` is interpreted as follows: The engine recognizes the `implicit` namespace and the `10nn` operator as an implicit relation that defines a virtual link between a subject and its ten most similar neighbors. This search is performed over the results of a derived relation defined by the `vec512` predicate, which relates document nodes to vectors of dimension 512.

■ **Listing 7** Hybrid SPARQL query combining symbolic filtering with implicit and derived relations.

```
PREFIX synth: <http://megras.org/synth#>

SELECT ?s ?o
WHERE {
  ?s synth:sel001 "true" .
  ?s <http://megras.org/implicit/10nn/vec512> ?o .
}
```

The results of this balanced benchmark, performed with a full system restart between query types, are summarized in Table 3. In this configuration, symbolic-only queries achieve a mean latency of 24.00 ms, while standalone 10-NN vector searches average 51.30 ms. The integrated hybrid query yields a mean latency of 57.60 ms, resulting in an additive join overhead of only 6.30 ms over the standalone vector search.

6.3.2 The Pushdown Effect

To evaluate the impact of early binding, we conducted an experiment measuring query latency across varying levels of filter selectivity: the fraction of the dataset that satisfies a symbolic predicate. A lower selectivity (e.g., 0.1%) indicates a highly restrictive filter, whereas higher selectivity (e.g., 50%) implies a larger candidate set for subsequent sub-symbolic operations. The thresholds for the selectivity were set as follows: $sel \in 0.001, 0.01, 0.1, 0.5$.

3:18 A Hybrid Vector-Symbolic Data Model for Multimodal Knowledge Graphs

■ **Table 3** Performance comparison with a consistent result set size ($n=10$) for all query types. All timing values (Mean, Cold Start, Min, Max, Median, and Std. Dev.) are reported in milliseconds (ms).

Query Type	Mean	Cold Start	Min	Max	Median	Std. Dev.
Symbolic-only	24.00	137	20	33	23.00	3.46
Vector-only	51.30	203	42	66	50.50	7.86
Hybrid	57.60	196	51	68	56.00	5.82

■ **Table 4** Detailed performance comparison across different selectivity levels for the default Jena engine vs. our optimized **BatchingQueryEngine**. All values are reported in milliseconds (ms) unless otherwise noted.

Selectivity	Engine	Mean	Speedup	Cold Start	Min	Max	Median	Std. Dev.
0.1%	Default	154.80	2.19×	244.00	115.00	209.00	150.00	26.13
	Batching	70.70		5,737.00	61.00	77.00	71.50	4.52
1.0%	Default	1,134.20	2.84×	1,753.00	945.00	1,429.00	1,150.50	133.63
	Batching	398.90		1,800.00	364.00	451.00	396.50	23.94
10.0%	Default	9,566.60	2.80×	11,505.00	8,885.00	10,473.00	9,474.00	503.36
	Batching	3,421.40		9,289.00	3,091.00	4,207.00	3,305.00	295.65
50.0%	Default	45,918.70	2.70×	53,800.00	43,475.00	50,750.00	45,413.50	2,068.80
	Batching	16,990.40		31,564.00	15,570.00	17,957.00	17,049.50	752.25

The benchmark used a hybrid query pattern designed to induce the $N + 1$ database call problem. It first filters nodes by a symbolic property, then retrieves the 10-nearest neighbors for each remaining node using a 512-dimensional vector. The representative SPARQL query for the 0.1% selectivity level is shown below:

■ **Listing 8** Example query designed to induce the $N + 1$ problem to demonstrate the pushdown effect.

```
PREFIX synth: <http://megras.org/synth#>

SELECT ?s ?neighbor ?prop
WHERE {
  ?s synth:sel001 "true" .
  ?s <http://megras.org/implicit/10nn/vec512> ?neighbor .
  ?s synth:prop_0 ?prop .
}
```

We compared Apache Jena’s default execution strategy against *MeGraS*’s **BatchingQueryEngine**. As detailed in Table 4, the **BatchingQueryEngine** provides an average speedup of factor 2.63, computed as the arithmetic mean of the per-selectivity speedup factors reported in the table. While Jena’s engine exhibits a linear scaling factor of around 91,711 ms per unit of selectivity, the optimized engine reduces this to 33,907 ms.

The performance disparity corresponds to the number of discrete database calls initiated by the query engine. The default engine executes $N + 1$ calls, where N is the number of nodes satisfying the initial symbolic filter. In contrast, the **BatchingQueryEngine** groups these operations into 2 or 3 bulk calls. Across all selectivity levels, the **BatchingQueryEngine** also recorded lower standard deviations.

■ **Table 5** Performance comparison across increasing graph volumes (selectivity=1%). All values are reported in milliseconds (ms) unless otherwise noted.

Volume	Engine	Mean	Speedup	Cold Start	Min	Max	Median	Std. Dev.
10^5	Default	1,235.00	3.05×	1,684.00	1,185.00	1,327.00	1,235.00	44.82
	Batching	404.50		1,276.00	352.00	464.00	391.50	32.18
10^6	Default	1,260.40	2.64×	1,745.00	1,192.00	1,348.00	1,257.00	50.31
	Batching	476.60		2,998.00	388.00	584.00	488.50	69.68
10^7	Default	1,297.20	2.39×	1,840.00	1,245.00	1,364.00	1,293.00	36.41
	Batching	543.50		7,700.00	482.00	614.00	544.50	42.77

6.3.3 Retrieval Scalability

To evaluate the `BatchingQueryEngine`'s performance under varying data complexities, we isolate two primary dimensions of scalability: the number of triples within the graph and the dimensionality of the vector embeddings. By fixing selectivity and query patterns, we measure the overhead introduced by the storage layer and the indexing structures to identify the system's operational bounds.

6.3.3.1 Scaling with Graph Volume

To assess the scalability of the *MeGraS* architecture, we evaluated query performance across three synthetic datasets of increasing magnitude: 10^5 , 10^6 , and 10^7 triples. We fixed the selectivity level to 1% (targeting 10 multimedia nodes) across all volumes, keeping the result count constant at 100 (since the 10-nearest neighbors are retrieved for each image node). Below is the hybrid query utilized:

■ **Listing 9** The hybrid query used for volume scalability benchmarking.

```
PREFIX synth: <http://megras.org/synth#>

SELECT ?s ?neighbor ?prop
WHERE {
  ?s synth:sel01 "true" .
  ?s <http://megras.org/implicit/10nn/vec512> ?neighbor .
  ?s synth:prop_0 ?prop .
}
```

As shown in Table 5, the `BatchingQueryEngine` consistently outperforms the default Jena execution strategy across all volume tiers. While both engines exhibit increased latency as the graph size grows, the `BatchingQueryEngine` maintains an average speed-up factor of 2.69, the arithmetic mean of the per-volume speedups reported in Table 5 (3.05×, 2.64×, and 2.39×). At the 10^7 triple mark, the optimized engine achieves a mean latency of 543.50 ms, whereas the default engine requires 1,297.20 ms to resolve the same query. The `BatchingQueryEngine` keeps the number of sub-symbolic resolution calls near-constant.

6.3.3.2 Scaling with Vector Dimensionality

The second component of our scalability analysis examines the impact of vector dimensionality on query performance. This experiment measures the latency of the 10-nearest-neighbor operation across dimensions ranging from 256 to 1024. We limited our testing to a maximum dimensionality of 1,024 due to the underlying `pgvector`'s HNSW implementation limits. The representative SPARQL pattern for the 1,024-dimensional vector predicate is shown below:

■ **Listing 10** Representative hybrid query for evaluating scalability across vector dimensions.

```
PREFIX synth: <http://megras.org/synth#>

SELECT ?s ?neighbor ?prop
WHERE {
  ?s synth:sel001 "true" .
  ?s <http://megras.org/implicit/10nn/vec1024> ?neighbor .
  ?s synth:prop_0 ?prop .
}
```

As illustrated in Table 6, the `BatchingQueryEngine` recorded a mean latency increase of 8.9 ms when moving from 256 to 1024 dimensions (from 52.50 ms to 61.40 ms). The standard deviation ranged from 4.39 ms to 5.14 ms across all tiers. The results show a scaling factor of approximately 0.012 ms per additional dimension.

■ **Table 6** Performance metrics of the `BatchingQueryEngine` across increasing vector dimensions. All values are reported in milliseconds (ms).

Dimensions	Mean	Cold Start	Min	Max	Median	Std. Dev.
256	52.50	202.00	43.00	58.00	54.00	4.82
512	55.10	221.00	46.00	60.00	56.00	4.76
768	57.50	231.00	51.00	65.00	56.50	4.39
1,024	61.40	261.00	55.00	73.00	59.50	5.14

6.3.4 Data Ingestion Throughput

To assess the performance of *MeGraS* under heavy workloads, we conducted a scalability analysis focusing on data ingestion throughput. The experiment measured the ingestion rate of multimodal triples as the total volume of the persistence layer increased toward a 10^7 quad limit. For this benchmark, we used the **MeGraS-SYNTH** dataset processed in batches of 10,000 items each.

The benchmark was orchestrated using a custom Kotlin routine that manages the data stream from the TSV source to the persistence store. Specifically, the routine reads the **MeGraS-SYNTH** TSV dump line by line, parses each line into typed `QuadValues` (distinguishing URIs, literals, and vector values), and accumulates the resulting quads into set-based batches of 10,000, deduplicating within each batch. Each set is handed to the persistence layer through the same `QuadSet` interface used during query execution: the store first resolves all distinct values to numeric identifiers via its in-memory dictionary caches, inserting new dictionary entries where necessary, with vector values routed to their dimension- and precision-specific tables (see Section 5); it then computes a content hash per quad, filters out statements already present in the store, and writes the remainder as multi-row bulk inserts. HNSW indices are created together with each vector table

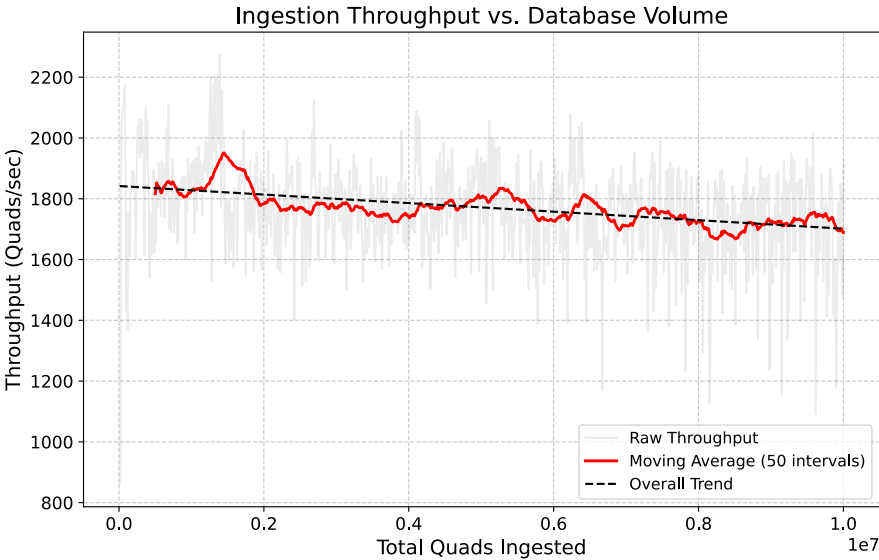
■ **Table 7** Ingestion performance analysis for the MeGraS-SYNTH dataset, consisting of 10^7 quads.

(a) Overall statistics of the dataset ingestion.

Metric	Value
Total Quads	10,001,611
Total Time (s)	5,712.00
Mean Rate (Q/s)	1,766.42
Median Rate (Q/s)	1,779.36
Peak Rate (Q/s)	2,277.39
Standard Deviation	154.04

(b) Throughput at scale milestones during the dataset ingestion.

Volume	Rate (Q/s)	Time (s)
0.01M	857.41	11.7
0.10M	1,432.87	64.0
1.00M	1,916.08	559.1
2.50M	1,917.18	1,384.0
5.00M	1,935.73	2,802.1
7.50M	1,743.07	4,232.4
10.00M	1,830.83	5,711.2



■ **Figure 2** Throughput analysis of the MeGraS persistence layer ingestion. The initial “warm-up” phase is followed by a stable rate of around 1,800 items per second. This stability is maintained across the entire 10-million-triple workload.

upon its first instantiation and are maintained incrementally thereafter; the reported throughput therefore includes dictionary maintenance, global deduplication, and index-update costs. Before the execution, the database was cleared to ensure a “cold-start” state. The system recorded the elapsed time and throughput rate (items per second) at every 10,000-item interval.

As summarized in Table 7a, the ingestion of the full 10-million-triple dataset was completed in 5,712 seconds, achieving a mean rate of 1,766.42 quads per second (Q/s). Table 7b shows that the rate began at 857.41 Q/s and reached a peak of 2,277.39 Q/s. The median throughput was 1,779.36 Q/s, with a standard deviation of 154.04 Q/s. At the 10^7 quad limit, the ingestion rate was 1,830.83 Q/s. Figure 2 visualizes the throughput per interval, the 50-moving average, as well as the linear overall trend.

■ **Table 8** Benchmark results for dynamic vs. materialized resolution of 1,000 CLIP embeddings. All values are reported in milliseconds (ms) unless otherwise noted.

Metric	Materialized	Dynamic
Mean Latency	1,314.30	52,596.10
Median Latency	1,312.50	51,264.50
Min Latency	1,258.00	48,281.00
Max Latency	1,379.00	59,180.00
Std. Deviation	45.22	3,380.74
Throughput (img/sec)	760.85	19.01

6.3.5 Advanced Features: Dynamic Knowledge Generation

Finally, we evaluate the computational cost of resolving relations that are not yet materialized: on-the-fly feature extraction. The term “dynamic knowledge generation” refers to the on-demand resolution of *derived relations* (Definition 3.11): when a query requests `derived:clipEmbedding` for a subject whose embedding does not yet exist in the graph, the engine invokes the external feature-extraction service on the already-ingested image at query time and persists the resulting triple. The “generated knowledge” thus consists of new, durable graph statements rather than ephemeral computation results. Accordingly, the *dynamic* state below measures the first such resolution for all 1,000 images, while the *materialized* state measures repeated retrieval of the persisted triples. To quantify this “inference tax,” we compare the initial resolution of the `megras:clipEmbedding` predicate (the dynamic state) against subsequent retrieval from the store (the materialized state) using the 1,000 images of the MeGraS-Synth dataset. The benchmark uses the following query to retrieve CLIP embeddings:

■ **Listing 11** SPARQL query that will retrieve relations from the graph or derive them, depending on their status in the graph.

```
PREFIX derived: <http://megras.org/derived/>

SELECT ?subject ?embedding
WHERE {
  ?subject derived:clipEmbedding ?embedding .
}
LIMIT 1000
```

Our benchmarks reveal a performance contrast, as summarized in Table 8: the materialized state achieves a mean retrieval latency of 1,314.30 ms, while the dynamic state results in a mean latency of 52,596.10 ms. This represents an overhead of approximately 3,900%. In the dynamic state, the throughput was recorded at 19.01 images per second, whereas the materialized approach achieved 760.85 images per second. Storing these 512-dimensional embeddings requires approximately 2 KB per image.

7 Discussion

The experimental evaluation of *MeGraS* in Section 6 provides a comprehensive view of the trade-offs and efficiencies inherent in hybrid multimodal knowledge graph architectures. By subjecting the system to both controlled synthetic benchmarks and the “sanity check” of a live retrieval

competition, we can characterize how specific design choices address the unique bottlenecks of sub-symbolic processing. The following synthesizes these findings to evaluate the system’s feasibility in real-world scenarios and its theoretical scaling limits under high-complexity workloads.

7.1 Real-World Feasibility

The results in Section 6.2 demonstrate that *MeGraS* effectively bridges the gap between high-dimensional vector search and symbolic metadata. The robustness of the **BatchingQueryEngine** is particularly evident in high-cardinality tasks (e.g., Listing 1), where the optimization of literal property retrieval prevents the serial bottlenecks common in traditional stores due to serial graph traversal patterns.

The low latency in temporal neighborhood retrieval effectively also supports the “walking through time” metaphor. Similarly, the performance of the text-search filters demonstrates the effectiveness of the underlying store’s indexing. By bridging symbolic metadata with text indices, the engine maintains a quick turnaround, even when processing string-heavy filters. Both of these aforementioned points are critical requirements for real-time tasks. Even in computationally intensive k -NN joins, the stable execution pipeline ensures that response times remain viable for real-time human-in-the-loop search by binding the vector similarity join early, preventing a nested-loop join that would otherwise incur one round-trip per bound subject.

Furthermore, the disparity between the observed cold-start and warm-start latencies reveals the efficiency of the underlying **PostgresStore** page-level caching. Specifically, the high-dimensional HNSW index segments that must be paged into memory to perform the k -NN search account for the initial 40% overhead in vector queries. However, for interactive retrieval, this trade-off is advantageous. By front-loading the cost of index loading, *MeGraS* enables fast subsequent executions: warm latencies remain well below 500 ms for the symbolic and text-filter queries (Listings 1–3), while the vector-join ADHOC query (Listing 4) stabilizes around two seconds; still comfortably within the interactive bounds of the three- to five-minute LSC task windows. We argue that this profile (a one-time paging cost followed by consistently fast responses) is more conducive to search success than a uniform but slower response time.

In conclusion, as evidenced by the latencies across the LSC ’25 tasks, *MeGraS* demonstrates significant robustness when processing heterogeneous datasets. The **BatchingQueryEngine** successfully navigated noisy OCR and skewed distributions without relational failure or the performance degradation typically associated with unoptimized hybrid joins. To place these results in the competitive context of the challenge itself: LifeGraph 5, which operated on an earlier *MeGraS* configuration using Jena’s default tuple-at-a-time execution, completed 9 of 18 tasks at LSC ’25 but was not competitive with the top-ranked, special-purpose retrieval systems, with query execution time identified as the dominant limiting factor [17]. The **BatchingQueryEngine** evaluated here removes precisely this bottleneck. Whether the resulting latencies translate into competitive challenge scores against predominantly non-SPARQL systems can only be established through future live participation, which we consider an important external validation of the multimodal-knowledge-graph approach for communities beyond the Semantic Web.

7.2 Performance Analysis

The “Cost of Hybridity” evaluation shows that the integrated hybrid query adds only 6.30 ms over the standalone vector search: a 12.3% increase relative to the vector-only baseline, and notably *less* than the sum of the two isolated components (75.30 ms). This demonstrates that the output volume is constrained, and the engine-level cost of coordinating symbolic and sub-symbolic operators is negligible. Furthermore, the cold-start latencies provide evidence of the engine’s

efficiency in managing multimodal indices under load. The hybrid query’s cold start (196 ms) is nearly identical to the vector-only cold start (203 ms), despite the added complexity of the symbolic filter. This indicates that the **BatchingQueryEngine** effectively parallelizes the initial index paging, maintaining high performance even during the first execution after a system reset.

Building upon this internal coordination efficiency, the experimental results for the “Pushdown Effect” illustrate the architectural necessity of early binding in multimodal contexts, reducing execution complexity from a linear $N + 1$ growth in database calls to a near-constant number of bulk interactions. The performance gap widens as the number of candidate nodes grows (high selectivity), confirming that our batching approach effectively mitigates the $N + 1$ problem. Furthermore, the substantial reduction in standard deviation points to higher system predictability, a vital trait for stable knowledge graph query engines. We argue that pushing symbolic filters down to the storage layer before invoking sub-symbolic vector operators is essential. Without this optimization, latency grows linearly with the number of candidate nodes bound by the symbolic filter.

The same experiments also delineate the conditions under which the batching approach is *less* advantageous. First, batching shifts cost toward the first execution: the cold-start latency of the **BatchingQueryEngine** substantially exceeds that of the default engine at low selectivity (5,737 ms vs. 244 ms at 0.1%, see Table 4) and at the largest graph volume (7,700 ms vs. 1,840 ms at 10^7 triples, cf. Table 5), because materializing intermediate binding sets forces a larger volume of index segments to be paged in at once. Workloads dominated by unique, never-repeated queries over cold data would therefore benefit considerably less from the proposed strategy. Second, for highly selective filters that bind only a handful of subjects, the absolute savings shrink (154.80 ms vs. 70.70 ms at 0.1%), as there are few round-trips left to amortize. Third, the threshold-based switch between IN-list and VALUES-join SQL translation (Section 4) relies on a fixed batch-size boundary of 2,000 tuned for our PostgreSQL configuration, which may be suboptimal on other backends or hardware.

These architectural advantages are further validated by scalability benchmarks that assess whether these benefits persist as the search space grows. The volume-scalability experiments indicate that the $N + 1$ problem’s overhead remains the dominant performance bottleneck regardless of the total triple count. Even as the cold-start time increases with larger index sizes, the batching approach effectively mitigates linear performance degradation. Within the single-node configuration and the volume range tested here (up to 10^7 triples), the batching approach keeps latency growth flat.

While graph volume affects retrieval, the choice of embedding model determines feature space dimensionality, which directly affects the computational cost of similarity metrics. Regarding vector dimensionality, the results indicate a relatively flat performance curve, suggesting that the **BatchingQueryEngine** scales gracefully with model complexity. The consistently low standard deviation validates the stability of the sub-symbolic resolution layer under varying computational loads. The low scaling factor (0.012 ms) suggests that the primary bottleneck remains the retrieval and initial processing of subjects rather than the mathematical complexity of high-dimensional distance calculations. This efficiency is achieved through an optimized indexing strategy that ensures sub-symbolic relations do not become the weak link as embedding models evolve toward higher dimensionalities.

Similar to query performance, the robustness of the underlying persistence layer is evidenced by data ingestion throughput, which followed an “inverse degradation” pattern. The rate accelerated as the PostgreSQL just-in-time compiler and B-Tree indices optimized for the workload, maintaining a median throughput consistent with multi-gigabyte workloads. As shown by the linear trend line in Figure 2, the system exhibits excellent stability with a variance of only 7.5%, indicating that

MeGraS can handle significant growth across the 10-million-quad ingestion measured here. The low standard deviation confirms that the tiered storage configuration effectively handles the I/O pressure of high-dimensional vector indexing.

Beyond the query engine’s static performance, the long-term viability of a multimodal store depends on how it manages the inherent computational cost of sub-symbolic inference. The performance gap between dynamic and materialized states represents a fundamental throughput bottleneck for non-materialized stores. While materialization incurs a storage cost (approx. 2 KB per image), this expenditure is negligible compared to the computational effort required to regenerate features for every request.

By materializing these relations, the system ensures that subsequent queries leverage indexed lookups rather than redundant computation. This strategy fundamentally alters the system’s scaling laws: in the dynamic state, query cost is bound to model complexity and hardware acceleration, resulting in a linear relationship between result set size and execution time. By contrast, persisting this knowledge shifts the cost to optimized indexing structures, allowing the system to maintain interactive speeds even as extraction functions grow in complexity.

Furthermore, this persistence model enables *MeGraS* to function as a cache for expensive sub-symbolic computations, ensuring that the high energy and hardware costs of model inference are incurred only once per relation per multimedia asset. The 40-fold increase in throughput for subsequent lookups is critical for real-world multimedia applications where users perform exploratory queries over the same high-dimensional features.

In summary, the performance analysis confirms that the *MeGraS* architecture successfully neutralizes the traditional overhead of multimodal retrieval. By combining batch-oriented query pushdown with a proactive materialization strategy, the system transforms computationally expensive sub-symbolic inferences into efficient relational operations. These results demonstrate that the proposed **BatchingQueryEngine** not only maintains high throughput during ingestion and retrieval but also provides a predictable, scalable foundation for the next generation of multimodal knowledge graphs that include high-dimensional vectors.

8 Limitations and Future Work

MeGraS’s architecture [14], while providing a unified framework for symbolic and sub-symbolic data, faces several limitations. The current implementation relies on the **pgvector** extension for its PostgreSQL backend, which restricts vector dimensionality to 2,000 when using HNSW indices. This may exclude high-dimensional embeddings from modern large-scale models. Additionally, while the system supports multiple backends, it currently lacks a distributed query execution layer, which limits horizontal scalability. A related challenge is that dynamic knowledge generation through non-materialized triples requires synchronous access to transformation functions, such as GPU-accelerated feature extraction, which can introduce latency bottlenecks for large-scale graphs. Furthermore, the heuristic pattern reordering used to address the $N + 1$ query problem is effective but does not yet incorporate a full cost-based optimizer that utilizes cardinality estimates for complex hybrid joins. Finally, the system also exhibits a dependency on “early-binding” query execution. Moreover, the query planner relies on threshold-based heuristics for switching between execution strategies, which may not always yield optimal performance across diverse workloads.

The evaluation presented in this work is also subject to several empirical constraints. First, the current benchmarks rely on a single-node setup, which does not capture the network latency and synchronization overheads inherent in a distributed deployment. Also, while the 10^7 triple scale demonstrates robustness, it does not fully reflect the complexities of the “open world” which brings issues like schema heterogeneity or data quality problems. Whether the integration of embeddings

into the graph can itself mitigate such heterogeneity (e.g., by matching schema-divergent entities through vector similarity where symbolic alignment fails) is an interesting open question; we expect this to help for similarity-based retrieval but not for queries that depend on exact symbolic constraints, and a systematic investigation remains future work. Relatedly, *MeGraS-SYNTH* inflates 1,000 seed images with synthetic noise properties and calibrated selectivity predicates; this design deliberately trades distributional realism for precise control over selectivity and dimensionality, so conclusions drawn from it concern engine behavior under controlled conditions, with the LSC evaluation serving as the realistic complement. Furthermore, the performance gains observed for materialized resolution are highly dependent on the specific hardware used, particularly the I/O throughput of the storage medium and the available GPU memory for the initial extraction phase. Our tests primarily focused on CLIP-based embeddings, and the performance characteristics may vary significantly when using more computationally intensive models or different types of embeddings. Likewise, because there are no standardized multimodal SPARQL benchmarks in the community, it is difficult to perform a direct head-to-head comparison with other emerging hybrid stores under identical conditions. Finally, while the current implementation leverages the HNSW index within the *pgvector* extension to achieve low-latency retrieval, this approach introduces significant memory overhead.

Meanwhile, several research directions warrant further investigation to advance *MeGraS* beyond its current form as a research prototype. One avenue is to integrate cost-based index selection and index-aware query planning directly into the SPARQL algebra, enabling the optimizer to automatically choose between indices and sequential scan strategies based on selectivity estimates for k -NN operations on massive datasets. There is also significant potential in developing learned index structures that adaptively partition the vector space based on symbolic graph context, such as partitioning image embeddings by their respective class URIs. Building streaming ingestion pipelines for the *Unified Multimedia Segmentation Model* [19] would enable *MeGraS* to support live multimedia data and update graphs in real time as new segments are detected. We furthermore aim to formalize an extended SPARQL syntax that supports native vector arithmetic and manifold projections, allowing for more complex sub-symbolic reasoning within the query language. Future work may likewise explore integrating disk-based approximate nearest-neighbor structures or hybrid memory-disk caching strategies to mitigate the HNSW index’s hardware dependencies. Finally, investigating “late-fusion” optimization strategies within the unified model could provide a fallback mechanism for queries where the symbolic metadata is insufficient to prune the search space before vector similarity calculations.

9 Conclusions

In this article, we addressed the fundamental architectural and representational “schism” [12] that has historically separated symbolic knowledge graphs from sub-symbolic multimodal data. By formalizing the *MediaGraph* Data Model [18], we demonstrated that high-dimensional vectors and multimedia documents are not merely peripheral attributes but can be integrated as first-class citizens within a unified RDF-based graph structure. This representational shift, which elevates vectors from opaque literals to addressable nodes, represents a significant departure from traditional decoupled architectures. Our work proves that “early-binding” query execution is not only theoretically sound but practically superior, as it allows the SPARQL optimizer to perform joint evaluation over structural relationships and perceptual similarity within a single algebraic framework. By treating the k -NN operation as a graph-native primitive rather than an external filter, we provide a mathematical foundation for cross-modal retrieval that maintains the formal rigor of the Semantic Web [1].

The implementation and evaluation of *MeGraS* [14], our open-source multimodal knowledge graph store and query engine, provide concrete answers to the scalability concerns often associated with hybrid stores. Our experimental results show that the batching execution strategy can effectively bridge the performance gap between specialized vector databases and general-purpose graph stores. Specifically, our ability to mitigate the long-standing $N + 1$ query problem ensures that complex multimodal retrieval remains performant even as the volume of data scales to 10^7 triples. The performance benchmarks demonstrate that the overhead of maintaining a unified model is significantly outweighed by reduced data movement and the ability to perform complex joins at the storage layer. Furthermore, by integrating the *Unified Multimedia Segmentation Model* [19], we include a framework that preserves media granularity via URI-based segments rather than external pointers. This ensures that spatiotemporal querying and content-aware filtering are no longer disparate processes but are instead interleaved operations that benefit from the rich context of a knowledge graph.

Ultimately, the contributions presented in this article provide a foundational infrastructure for the next generation of multimodal knowledge graphs. We have moved beyond simple metadata storage toward a system that can interpret the internal structure of media assets in relation to global knowledge. By attempting to unify the symbolic and sub-symbolic domains, we offer a pathway toward more expressive retrieval strategies that can navigate the increasing heterogeneity of digital media without sacrificing interoperability. As the explosion of multimodal data continues, the principles embodied in *MeGraS* (unification, granularity, and “early-binding” optimization) will be essential for developing systems that can truly understand and retrieve information across the full spectrum of human-generated data.

References

- 1 Tim Berners-Lee, James Hendler, Lassila, and Ora. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
- 2 Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 2787–2795, 2013. URL: <https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html>.
- 3 Ralph Gasser, Luca Rossetto, Silvan Heller, and Heiko Schuldt. Cottontail DB: an open source database system for multimedia retrieval and analysis. In Chang Wen Chen, Rita Cucchiara, Xian-Sheng Hua, Guo-Jun Qi, Elisa Ricci, Zhengyou Zhang, and Roger Zimmermann, editors, *MM '20: The 28th ACM International Conference on Multimedia, Virtual Event / Seattle, WA, USA, October 12-16, 2020*, pages 4465–4468. ACM, 2020. doi:10.1145/3394171.3414538.
- 4 Cathal Gurrin, Liting Zhou, Graham Healy, Alie Tran, Luca Rossetto, Werner Bailer, Duc-Tien Dang-Nguyen, Steve Hodges, Björn Þór Jónsson, Minh-Triet Tran, and Klaus Schöffmann. Introduction to the 8th annual lifelog search challenge, lsc'25. In *Proceedings of the 2025 International Conference on Multimedia Retrieval, ICMR '25*, pages 2143–2144, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3731715.3734579.
- 5 Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. *Knowledge Graphs. Synthesis Lectures on Data, Semantics, and Knowledge*. Morgan & Claypool Publishers, 2021. doi:10.2200/S01125ED1V01Y202109DSK022.
- 6 Thomas Kurz, Sebastian Schaffert, Kai Schlegel, Florian Stegmaier, and Harald Kosch. SPARQL-MM - extending SPARQL to media fragments. In Valentina Presutti, Eva Blomqvist, Raphaël Troncy, Harald Sack, Ioannis Papadakis, and Anna Tordai, editors, *The Semantic Web: ESWC 2014 Satellite Events - ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, volume 8798 of *Lecture Notes in Computer Science*, pages 236–240. Springer, 2014. doi:10.1007/978-3-319-11955-7_26.
- 7 Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In David J. Fleet,

- Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V*, volume 8693 of *Lecture Notes in Computer Science*, pages 740–755. Springer, 2014. doi:10.1007/978-3-319-10602-1_48.
- 8 Lihui Liu, Zihao Wang, and Hanghang Tong. Neural-symbolic reasoning over knowledge graphs: A survey from a query perspective. *SIGKDD Explor.*, 27(1):124–136, 2025. doi:10.1145/3748239.3748249.
 - 9 Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020. doi:10.1109/TPAMI.2018.2889473.
 - 10 Sudip Mittal, Anupam Joshi, and Tim Finin. Thinking, fast and slow: Combining vector spaces and knowledge graphs. *CoRR*, abs/1708.03310, 2017. arXiv:1708.03310.
 - 11 Andriy Nikolov, Peter Haase, Daniel M. Herzig, Johannes Trame, and Artem Kozlov. Combining rdf graph data and embedding models for an augmented knowledge graph. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 977–980, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee. doi:10.1145/3184558.3191527.
 - 12 André Platzer. Intersymbolic AI - interlinking symbolic AI and subsymbolic AI. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part IV*, volume 15222 of *Lecture Notes in Computer Science*, pages 162–180. Springer, 2024. doi:10.1007/978-3-031-75387-9_11.
 - 13 Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021. URL: <http://proceedings.mlr.press/v139/radford21a.html>.
 - 14 Luca Rossetto and Florian Ruosch. MeGraS: An Open-Source Store for Multimodal Knowledge Graphs. In *Proceedings of the 33rd ACM International Conference on Multimedia (MM '25)*, MM '25, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3746027.3756872.
 - 15 Luca Rossetto, Florian Ruosch, and Jan Willi. MediaGraphOrg/MeGraS. Software, swhId: swh:1:dir:b0d522ad9603f43d5135aa393c24d74d7ea8a3d7 (visited on 2026-09-28). URL: <https://github.com/MediaGraphOrg/MeGraS>, doi:10.4230/artifacts.28092.
 - 16 Florian Ruosch and Luca Rossetto. MediaGraphOrg/MeGraS-SYNTH. Dataset, swhId: swh:1:dir:98b200e81f058bd739629f0f2dca77cb7ea523e6 (visited on 2026-09-28). URL: <https://github.com/MediaGraphOrg/MeGraS-SYNTH>, doi:10.4230/artifacts.28093.
 - 17 Florian Ruosch and Luca Rossetto. A SPARQL in the dark: Shining a light on multimodal lifelogs with LifeGraph 5. In *Proceedings of the 8th Annual ACM Workshop on the Lifelog Search Challenge (LSC '25)*, LSC '25, New York, NY, USA, 2025. ACM. doi:10.1145/3729459.3748694.
 - 18 Florian Ruosch and Luca Rossetto. Mediagraph: A content-aware data model and query framework for multimodal knowledge graphs, 2026. arXiv:2608.18895.
 - 19 Jan Willi, Abraham Bernstein, and Luca Rossetto. Unified multimedia segmentation - A comprehensive model for uri-based media segment representation. *TGDK*, 2(3):1:1–1:34, 2024. doi:10.4230/TGDK.2.3.1.
 - 20 Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. arXiv:1412.6575.