An LLM-Guided Query-Aware Inference System for GNN Models on Large Knowledge Graphs

Waleed Afandi
Concordia University

Hussein Abdallah
Concordia University

Ashraf Aboulnaga
University of Texas at Arlington

Essam Mansour
Concordia University

***Abstract*—Efficient inference for graph neural networks (GNNs) on large knowledge graphs (KGs) is essential for many real-world applications. GNN inference queries are computationally expensive and vary in complexity, as each involves a different number of target nodes linked to subgraphs of diverse densities and structures. Existing acceleration methods, such as pruning, quantization, and knowledge distillation, instantiate smaller models but do not adapt them to the *structure or semantics* of individual queries. They also store models as monolithic files that must be fully loaded, and miss the opportunity to retrieve only the neighboring nodes and corresponding model components that are semantically relevant to the target nodes. These limitations lead to excessive data loading and redundant computation on large KGs. This paper presents KG-WISE, a task-driven inference paradigm for large KGs. KG-WISE decomposes trained GNN models into fine-grained components that can be partially loaded based on the structure of the queried subgraph. It employs large language models (LLMs) to generate reusable query templates that extract semantically relevant subgraphs for each task, enabling query-aware and compact model instantiation. We evaluate KG-WISE on six large KGs with up to 42 million nodes and 166 million edges. KG-WISE achieves up to 28× faster inference and 98% lower memory usage than state-of-the-art systems while maintaining or improving accuracy across both commercial and open-weight LLMs.**

I. Introduction

Graph neural networks (GNNs) on knowledge graphs (KGs) have proven effective in many applications [1], such as recommendation [2], drug discovery [3], anomaly detection [4], and fraud detection [5]. An *inference query* on a KG uses a trained GNN model to predict missing information (e.g., link prediction or node classification) for a set of *target nodes* (TN) [6]. Efficient inference is essential for deploying GNNs on large KGs. The cost depends on the density and structure of the subgraphs around the target nodes. During inference, the system loads large data from disk to main memory and GPU memory, including the entire KG adjacency, model parameters, and node embeddings. It then performs dense tensor aggregation and message passing for each node, as shown in Figure 1. This process is computationally expensive and scales poorly as the KG grows. **This raises a central research problem: how can we perform scalable and adaptive GNN inference that tailors computation and data loading to the structure and semantics of each inference query on a large KG?**

Prior work on GNN inference acceleration focuses on converting trained models into smaller or faster versions for deployment [7]. Typical methods include pruning [8], quantization [9], and knowledge distillation [10]. These approaches

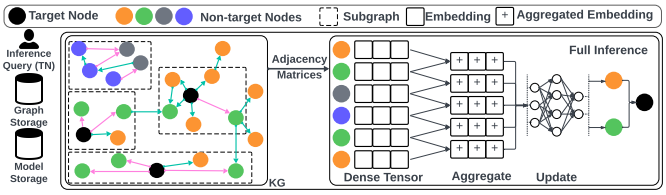

Fig. 1. A GNN inference query for target nodes (TN) loads the KG's adjacency, model, and embedding matrices from storage, then performs message passing and embedding updates to produce predictions. This process is resource-intensive and scales poorly with the size of these matrices in KGs.

mainly reduce model weights but ignore the cost of handling precomputed embeddings for non-target nodes, which must still be loaded during inference. Most systems store trained GNNs as monolithic files, forcing full model loading even when only a small part is relevant to the query. Another line of work performs inference on fixed L-hop neighborhoods around each target node [6]. While this reduces computation, it treats all queries uniformly and overlooks semantic and structural differences among them. Hence, these methods fail to adapt inference to the query context and often load less relevant neighbors, leading to redundant computation on large heterogeneous KGs. As a result, existing accelerators work well on small graphs but scale poorly in realistic KG settings. They miss the opportunity to adapt computation and data loading to each query by instantiating compact and query-specific models. Achieving this goal requires a system that performs scalable and adaptive inference tailored to the structure and semantics of each query.

GNN *training* accelerators improve scalability by sampling smaller subgraphs for mini-batch training. Methods such as random walk sampling (GraphSAGE [11]), node-importance sampling (ShaDowGNN [12] and IBMB [13]), and task-oriented sampling (KG-TOSA [14]) reduce training time and memory usage. However, these techniques are designed for training, not inference. At inference time, the full model and its complete training graph must still be loaded, regardless of the query's size or structure. They also assume fixed neighborhoods and ignore variation in query semantics and subgraph density. Thus, they fail to tailor computation and data loading to the structure and *semantics* of the GNN task on a large KG. For instance, KG-TOSA misses this opportunity by using fixed graph patterns that retrieve neighboring nodes without considering their semantic relevance to the target task.

This paper presents KG-WISE, a query-aware inference system that overcomes the limitations of existing GNN

inference approaches on large KGs. KG-WISE adapts the trained GNN model to the structure and semantics of each task through three key steps. First, during training, KG-WISE uses an LLM-based method to analyze the GNN task description and its target nodes (TN). The LLM identifies relevant entities and predicates in the KG schema and generates a SPARQL query template to retrieve the corresponding subgraph. This subgraph, significantly smaller than the full KG, is then used to train the model. After training, KG-WISE decomposes the model into modular components, weights and node embeddings, and stores them in a key-value store for partial loading. Second, during inference, KG-WISE reuses the stored query template to fetch a semantic subgraph relevant to the current inference query. It then instantiates a compact query-specific model by loading only the components associated with that subgraph. This design avoids redundant data movement and computation while preserving prediction accuracy.Third, KG-WISE dynamically selects between sparse or dense tensor aggregation based on the structure of the retrieved subgraph to further improve efficiency. Together, these techniques enable scalable, query-aware GNN inference that reduces memory and computation costs while maintaining model accuracy.

We evaluate KG-WISE on the KGBen benchmark [14], which includes KGs with up to 42M nodes and 166M edges across six domains (e.g., DBLP [15], MAG [16], YAGO4 [17], and WikiKG [18]). Tasks cover node classification and link prediction with 100–1600 target nodes to model realistic inference workloads. We compare against inference accelerators GCNP [8], Degree-Quant [9], and GKD [10] (when scalable), and training methods GraphSAINT [19], MorsE [20], IBMB [13], and KG-TOSA [14]. KG-WISE matches or improves accuracy while reducing inference time by up to 28× and memory by up to 98%. Ablations isolate the gains from LLM-guided subgraphs and partial model loading; query-size, scalability, and CPU/GPU tests show consistent improvements with low preprocessing overhead. We also evaluated KG-WISE with both commercial (Gemini, GPT4/5) and open-weight (GPT-oss, Qwen, DeepSeek) LLMs, observing comparable accuracy and efficiency across all models.

In summary, the main contributions of this paper are:

- The first end-to-end system for scalable GNN storage and inference on large KGs; Section III.
- A fine-grained decomposition and storage mechanism for GNN models on KGs that enables *partial model loading* with minimal preprocessing overhead; Section IV.
- A query-aware inference approach that uses LLM-guided query templates to extract semantically relevant subgraphs and instantiate compact models with negligible retrieval and loading cost; Section V.
- A comprehensive evaluation on large KGs showing that KG-WISE reduces inference time by up to 28× and memory by up to 98% while maintaining or improving accuracy. The results are consistent across commercial and open-weight LLMs, demonstrating that KG-WISE's performance is not tied to any specific model; Section VI.

II. Background and Related Work

A. Training GNNs on KGs

Training a GNN model on a KG takes the graph structure and its node features as input. The node features are used to initialize node embeddings. If node features are missing, node embeddings are initialized randomly. The training process generates node representations for downstream tasks along with the model parameters. Unlike GNN methods designed for homogeneous graphs, GNN methods for heterogeneous graphs, such as KGs, extend traditional approaches to support diverse node and edge types. These methods utilize multi-layer architectures, such as Relational Graph Convolutional Networks (RGCNs) [21] and often adopt sampling-based mini-batch training [19], [12], [20]. For instance, RGCN layers aggregate embeddings from neighboring nodes based on specific relation types. The hidden embedding of a node i at layer $l+1$ is computed as:

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in N_i^r}\frac{1}{c_{i,r}}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right) \quad (1)$$

$h_i^{(l+1)}$ is the updated embedding of node i, σ is an activation function, N_i^r is the set of neighbors of node i under relation r, $c_{i,r}$ is a normalization constant (e.g., $c_{i,r} = |N_i^r|$), $W_r^{(l)}$ is the weight matrix for relation r at layer l, and $W_0^{(l)}$ is the layer-specific weight matrix for self-loops. Training on large KGs is computationally expensive due to the need to aggregate messages from multi-hop neighbors, where nodes are connected through multiple edges of specific types ($\mathcal{R}$). This becomes more demanding in deep GNNs as information propagates through many layers, especially with a large $|\mathcal{R}|$ [14]. The computational overhead increases due to the size and heterogeneity of KGs, since training iterates over all the relations $\mathcal{R}$. The size of nodes and edges affects significantly the overall computations, as illustrated in Equation 1.

Mini-batch training scales GNNs by sampling subgraphs to reduce memory usage during training. Various RGCN-based methods employ different sampling strategies: GraphSAINT [19] uses random walk sampling, IBMB [13] applies personalized PageRank, Shadow-GNN [12] adopts node-importance sampling, MorsE [20] employs structure-aware sampling, and KG-TOSA [14] performs task-oriented sampling. However, these methods still require loading the entire trained model and the full graph at inference time (Figure 1).

B. Traditional GNN Inference Pipeline

The typical inference pipeline in GNN methods begins with an inference query $\mathcal{IQ}(TN)$ that requests predictions (e.g., Node Classification (NC) or Link Prediction (LP)) for a set of target nodes in a KG. The pipeline involves loading the entire KG and trained GNN model into memory, followed by GNN aggregation over dense node embeddings across multiple layers, as illustrated in Figure 1 and Equation 1. The inference pipeline performs a forward pass on a dense graph, where N is the number of nodes, L is the number of layers, and F

TABLE I
A COMPARATIVE ANALYSIS OF EXISTING AND PROPOSED INFERENCE ACCELERATION TECHNIQUES.

	Pruning (GCNP) [8]	**Quantization (DQ) [9]**	**Know. Distillation (GKD) [10]**	**Ours (KG-WISE)**
Optimization Overhead	Per layer Channel Pruning	Simulation of Quantization effect	Student Model Training	Decompose H into KV Store
Additional Training	✓	✓	✓	×
Storage Mechanism	Disk based single file	Disk based single file	Disk based single file	Disk based & Key-Value store
Partial Graph Loading	× (Full Graph)	× (Full Graph)	✓ (Randomly sparsed)	✓ (Query based)
Partial Embedding Loading	× (Full Graph)	× (Full Graph)	× (Full Graph)	✓ (1 hop)
Forward Pass	Full	Full	Random Sparse	Query Aware
Inference Time Complexity	$O(L'N^2F' + L'NF'^2)$	$O(LN^2F' + LNF'^2)$	$O(LN'^2F + LN'F^2)$	$O(LN_{SG}^2F_{SG} + LN_{SG}F_{SG}^2)$

is the average node embedding size per layer [22]. The time complexity is $O(LN^2F+LNF^2)$ and the space complexity is $O(N^2+LF^2+LNF)$. This process is memory-intensive and computationally costly, particularly for large graphs. It may also involve unnecessary computations, such as calculating embeddings for unreachable non-target nodes, which do not impact target node embeddings, as shown in Figure 1.

Full Inference vs. Batched Inference: In full inference, the entire graph is used for aggregation, where each node aggregates information from all its neighbors across the graph. Although accurate, this approach is computationally prohibitive for large graphs. In contrast, batched inference divides target nodes into smaller batches and processes them iteratively via subgraphs sampling. This approach reduces memory usage but may compromise aggregation accuracy as it limits access to a node's full neighborhood.

C. GNN Inference Acceleration Methods

Given a trained GNN model M, inference acceleration aims to derive a smaller and faster model $\widetilde{M}$ that preserves the accuracy of M. These accelerators typically require access to the graph and often involve retraining. The instantiated model $\widetilde{M}$ may change architecture, parameter precision, or weight representations. Existing techniques fall into three main categories: pruning, quantization, and knowledge distillation. Table I compares these methods with KG-WISE.

Homogeneous vs. Heterogeneous Graphs. Most existing accelerators are designed for homogeneous graphs, where all nodes share the same type and embeddings are generated on demand during inference. In heterogeneous graphs such as KGs, however, the model stores large precomputed embeddings for non-target nodes to capture semantic diversity. These embeddings often dominate model size and must still be fully loaded at inference time, even when only a small subset is relevant to the query. In the following, we review the main categories of GNN inference acceleration methods and discuss their general principles and limitations.

GNN Pruning: This approach aims to identify and prune specific weights W in the GNN model M without compromising its accuracy [8], [23]. The approach applies an additional step after the model M has been trained and involves per-layer channel pruning. This approach necessitates extra training to identify the weights to remove [8], [23]. The pruning reduces the model size by reducing the complexity of layer L' via its reduced hidden layer feature size F', which contributes to a smaller W. Instantiating the inference pipeline requires loading both the full model $\widetilde{M}$ and the complete graph. The pruned model $\widetilde{M}$ is stored on disk as a single file. In this approach, the inference process performs computation on the entire graph, regardless of the target node set, which is expensive and which KG-WISE aims to avoid.

GNN Quantization: The quantization approach reduces the numerical precision of the model parameters. It represents the hidden layer features with fewer bits, yielding an F' that contributes to faster matrix multiplications of W and a smaller version of the model, $\widetilde{M}$, while maintaining performance comparable to M [24], [9]. Like pruning, quantization requires additional training to instantiate $\widetilde{M}$. This training employs a Quantization Aware Training (QAT) framework, where the model M is trained as if its weights W are quantized. This prepares the model to handle the effects of quantization in its final deployment stage and adapt to quantization effects to mitigate accuracy loss. Quantization reduces model size by using smaller weights $\widetilde{W}$, but it does not prune parameters or layers. Both M and $\widetilde{M}$ models are stored as a single file on disk. As before, inference requires full loading of both the graph and model $\widetilde{M}$.

GNN Knowledge Distillation (KD): This approach focuses on encoding and transferring geometric information graph structure from a teacher GNN model to a student GNN model. The student model is designed to be smaller, with fewer N' nodes and E' edges to achieve faster inference and comparable model performance to the teacher [25], [26]. For instance, methods, such as GKD [10], train a teacher model first and then train a student model to replicate the teacher's geometric understanding. The student model is trained using a randomly sparse graph, which is generated by removing nodes and edges randomly from the KG.

Limitations of Existing Accelerators. Across pruning, quantization, and KD, the instantiated model $\widetilde{M}$ is always a *single static artifact* stored as one file. None of these methods decompose embeddings by node type or support partial loading. As a result, even inference over a small number of target nodes, common in real-world deployments [27], [6], forces full-model materialization, leading to high memory overhead and inefficient inference on KGs.

KG-WISE differs from traditional black-box accelerators that operate on arbitrary pre-trained models. It follows a task-driven inference paradigm where training and inference are aligned through semantic subgraph extraction. Only task-

relevant model components and embeddings are loaded into memory. This goes beyond standard sampling-based pipelines and enables more efficient inference.

III. KG-WISE System Overview

KG-WISE addresses the challenge of performing scalable and adaptive GNN inference that tailors computation and data loading to the structure and semantics of each query on large KGs. It introduces three key innovations: (1) an LLM-guided method that generates reusable SPARQL query templates for extracting semantically relevant subgraphs, (2) fine-grained model decomposition that enables partial model loading, and (3) query-aware model instantiation that adapts inference to each query's subgraph structure. Figure 2 presents the architecture of KG-WISE. The system consists of a *GNN Manager* and a *GNN Storage Manager*, supported by an RDF engine for knowledge graph storage and metadata management. We next describe each component following the system workflow.

LLM-guided Query Template and Training Phase: During training, KG-WISE learns a compact task-specific model from semantically relevant subgraphs instead of the full KG. A single task refers to a specific target node type, having specific related relations, and specific tails or labels. These subgraphs are retrieved via SPARQL queries, and we generate a query template for these queries, Q_T, using an LLM-guided procedure (Algorithm 1), *executed once before training*.

The LLM is prompted with both the task description and schema statistics. All prompts and queries used by KG-WISE are in the full version of this paper [28]. These statistics include triple-type frequencies and relation coverage, which implicitly capture schema density. This information is retained during schema pruning (Line 2) and reused when selecting basic graph patterns (BGPs) (Lines 3–4), guiding the LLM toward structurally relevant and well-populated relations. The verified BGPs are then compiled into a reusable query template Q_T (Lines 5–7), which is applied during both training and inference to extract a dense, task-aligned subgraph without re-invoking the LLM. Given schema-level BGPs $\mathcal{B} = \{(s_i, p_i, o_i)\}$ and a target node type VT, KG-WISE uses an LLM to generate a constrained SPARQL query template for task-relevant subgraph extraction:

$$Q_T = g(\mathcal{B}, VT, KG_{sc}) \tag{2}$$

where KG_{sc} is the KG schema and $g(\cdot)$ is a prompt-guided function. The prompt enforces schema-valid template population, correct edge directionality, and VT-rooted nested subqueries combined via `UNION`. Schema verification prevents hallucinated relations and ensures reproducibility.

The Training Manager executes Q_T on the RDF engine to retrieve the task subgraph SG and trains a GNN model M on it. SG is agnostic to the underlying GNN architecture and can be readily integrated with other GNN models to improve their training and inferencing quality. After training, the *Decomposition Manager* splits M into node embeddings and model parameters, stored in a key-value (KV) store for fine-grained access. Each embedding chunk is indexed by node type to allow selective retrieval during inference. The query template Q_T and model metadata are stored in the RDF engine for reuse.

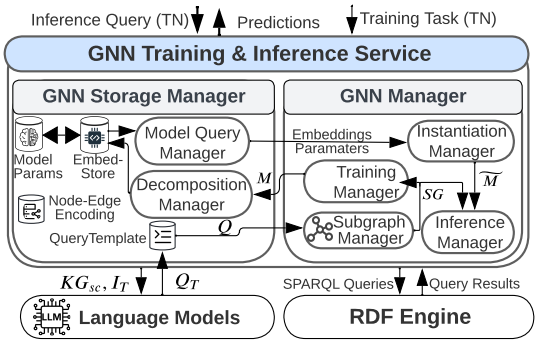

Fig. 2. KG-WISE orchestrates training and inference on large KGs through LLM-guided subgraph extraction, fine-grained model storage, and query-aware model instantiation.

Algorithm 1 LLM-Guided Subgraph Query Generation

Input: Knowledge Graph KG, Schema KG_{sc}, Number of Hops K, Task Instruction I_T, SPARQL Example Q_E
Output: Query Template Q_T
1: **function** SUBGRAPH_QUERY_GEN(KG_{sc}, K, I_T)
2: $I_{SF} \leftarrow$ suggestTaskFeatures(I_T)
3: $KG_{ps} \leftarrow$ pruneSchema(KG_{sc}, K, I_T)
4: $I_{BGP} \leftarrow$ mapToBGP(I_{SF}, KG_{ps})
5: $I_{BGP} \leftarrow$ verifyBGPs(I_{BGP}, KG_{ps})
6: $Q_T \leftarrow$ BGPToSPARQL(I_{BGP}, Q_E)
7: $Q_T \leftarrow$ verifySPARQL(Q_T, KG_{ps})
8: **return** Q_T

A Query-aware Inference Phase: At inference time, KG-WISE adapts computation and data loading to the semantics of each query. Given a query targeting nodes TN, the *Subgraph Manager* retrieves the stored template Q_T from the RDF engine and executes it to extract the semantic subgraph SG relevant to TN. No additional LLM calls are required. The *Instantiation Manager* collaborates with the *Model Query Manager* to fetch only the model components (weights W and embeddings $E(RN_M)$) needed for SG. These components are combined to instantiate a compact query-specific model $\widetilde{M}$. Depending on subgraph sparsity, the Instantiation Manager selects sparse or dense tensor aggregation to optimize computation. The *Inference Manager* then performs the forward pass on $\widetilde{M}$ to generate predictions for TN, achieving efficient and accurate inference with minimal memory use.

Storage and Metadata Management: KG-WISE employs a dual-store design for efficient data access. An RDF engine stores the KG and model metadata, including schema, task definitions, and query templates. This enables semantic subgraph retrieval through SPARQL queries using built-in RDF indices. A KV-store manages numerical tensors such as embeddings and model parameters, chunked and indexed by node type for partial loading. This separation of semantic metadata and numerical tensors allows KG-WISE to perform both graph retrieval and fine-grained model access efficiently.

Discussion: Unlike prior systems that load the full model and graph for each inference query, KG-WISE performs adaptive GNN inference by tailoring data loading and computation to

each query's structure and semantics. Its LLM-guided subgraph templates and fine-grained model decomposition enable partial model loading and compact model instantiation. While this mechanism also benefits training efficiency, its primary impact lies in reducing inference time and memory usage on large heterogeneous KGs.

IV. The KG-WISE Storage Mechanism

This section presents KG-WISE's model decomposition approach and its efficient storage and retrieval methods.

A. GNN Model Layout and Storage Mapping

ALL GNN models have a typical layout, shown in the top half of Figure 3, consisting of three core components: node/edge encodings, model parameters, and node embeddings, all linked to an encoded KG. Node/edge encodings are unique identifiers for graph elements. They help transform the inference-related subgraph into adjacency matrices that align the KG with the model. Each node has a unique identifier that remains consistent during both training and inference to ensure stable mapping. Each layer's weights are stored in the Parameter Store. Node embeddings capture the graph semantics of non-target nodes. These embeddings are learned during training and are stored as $m \times n$ arrays, where m is the number of nodes and n is the embedding size. Excessively increasing n leads to high memory usage.

KG-WISE decomposes the GNN model and stores its components in a scalable storage system, as shown in Figure 3. The Encoding Store maps graph elements, such as nodes and edges, to their respective encodings to enable efficient adjacency matrix construction. The Parameter Store manages the model's weights to ensure quick access during inference. Node embeddings are divided into chunks by node type, and these chunks are stored in a KV store, indexed by node type and node ID for efficient retrieval. This modular storage design supports random access to relevant data during inference, which helps reduce data loading and memory usage.

B. The Decomposition Algorithm

Algorithm 2 decomposes the trained GNN model and stores its components in a KV store. It first iterates over all node types, retrieves their embeddings (Lines 2–3), and chunks them by size and embedding dimension before writing them to disk (Line 4). As node encoding is already handled during training, no additional processing is needed when saving embeddings. Chunking divides large embeddings into smaller, manageable parts, improving memory efficiency and retrieval speed. For relation-specific weights, the algorithm traverses each model layer (Lines 5–6), extracts the corresponding weights (Line 7), and writes them to disk (Line 8). These relational weights are stored together as a single file for each layer.

The node embeddings are chunked and stored in a KV store, while the model parameters and weights are saved separately in the Parameter Store. The model parameters are much smaller than the embeddings. Hence, storing them separately is more efficient. This separation allows for independent and efficient access to node embeddings and parameters. KG-WISE supports parallel storage and retrieval operations, which is useful for large GNN models. As the graph size and embedding dimensions increase, the KV store provides scalable storage, which ensures fast processing and efficient retrieval. The decomposition algorithm integrates seamlessly with the KV store.

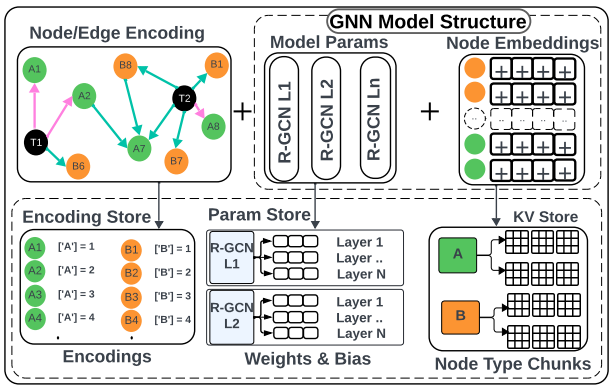

Fig. 3. The top half shows an encoded KG and a GNN trained on it. The GNN's main components are the learned parameters and **non-target node embeddings**. The bottom half illustrates how KG-WISE maps the encoding and decouples the GNN components. The node embeddings are stored in a key-value store as as row-wise chunks grouped by node type in a Zarr KV store.

Algorithm 2 GNN Model Decomposition

Input: *Model M; Chunk size c; Embedding dim e; Key-Value Store KVS*
Output: Decomposed Model

1: **function** DECOMPOSE(M, c, e, KVS)
2: **for** each vertex type $v_t \in M_{vertexTypes}$ **do**
3: $E \leftarrow M_{v_t}^{emb}$ ▷ Retrieve embeddings for vertex type v_t
4: KVS.write(E, c, e) ▷ Store embeddings in KVS in chunks
5: **for** each layer $l \in M_{layers}$ **do**
6: **for** each relation type $r_t \in M_l$ **do**
7: $W_{l,r} \leftarrow Weights(l, r)$
8: storeConvWeightsToDisc$(W_{l,r})$ ▷ Store layer weights to disc

C. Efficient Storage and Retrieval

Efficient Chunking and Indexing of Node Embeddings KG-WISE uses chunking to reduce I/O overhead during inference by loading only the embeddings of nodes that actually appear in the inference subgraph. Chunking is *row-wise*: each chunk contains full embedding vectors for a subset of nodes, not parts of a single embedding vector. This enables KG-WISE to retrieve embeddings for relevant non-target nodes by type and ID, rather than loading the entire embedding matrix. The chunk size is configurable and selected based on graph size and available memory. We discuss this trade-off and provide empirical guidance in the supplementary material.

Zarr KV Store We use Zarr [29] as the underlying KV store because it provides built-in block indexing over chunked arrays. Each node type is stored as a separate Zarr array, internally indexed by contiguous embedding blocks. During inference, the Model Query Manager fetches only the chunks containing the node IDs present in the extracted subgraph. This design differs from approximate nearest neighbor (ANN) indices [30], [31], as KG-WISE retrieves exact embeddings by node ID rather than similar embeddings by vector similarity

search. Zarr's chunk-level indexing ensures fast access without scanning unrelated embeddings. Each GNN task maintains its own Zarr instance, and updates are rare since embeddings only change when new neighbor nodes are introduced.We chose to use Zarr as our underlying storage format due to its superior support for multi-dimensional chunking and high-performance compression (e.g., Blosc). While PyG's standard *FeatureStore* provides a standardized API for GNN data access, it is fundamentally an abstraction layer that requires a storage backend. This organization ensures scalable storage, minimizes unnecessary loading, and supports efficient partial retrieval during query-aware inference.

V. The KG-WISE Query-Aware Inference Pipeline

This section presents our query-aware inference pipeline, which tailors computation and data loading to each query's structure and semantics. KG-WISE uses an LLM-generated SPARQL template (created once during training) to extract a relevant subgraph and instantiate a compact, query-specific model $\widetilde{M}$. As shown in Figure 4, only the necessary embeddings are loaded, enabling much smaller model instances. As a result, KG-WISE achieves up to 28× faster inference and 98% lower memory usage while maintaining comparable or better accuracy.

A. Extracting the Inference-related Subgraph

During inference, GNNs aggregate embeddings from the neighbors of each target node TN. In large KGs, many neighbors are unreachable or semantically irrelevant, and loading their embeddings adds computation and memory cost without improving accuracy. KG-WISE avoids this overhead by extracting a compact inference subgraph SG that contains only semantically relevant nodes within K hops of TN. The RDF engine executes a SPARQL query derived from the stored task template Q_T (Algorithm 3). Since the template is generated once during training, no LLM calls are needed at inference time. This ensures consistency across repeated queries and avoids runtime prompt overhead.

The retrieved triples are then deduplicated and encoded. Node, relation, and label encodings are preserved to match those used during training, ensuring a seamless mapping between SG and the stored embeddings. The final SG is exported as a list of triples and converted to a PyG-compatible format with the same node identifiers and types used during model training. Algorithm 3 shows this process. In Line 2, the template Q_T is instantiated with the target nodes TN. In Line 3, the concrete SPARQL query is executed to collect the triples. In Line 4, duplicates are removed. Lines 5–6 apply node and relation encodings, and Line 7 attaches label encodings. The fully encoded inference subgraph SG is returned in Line 8. This produces a dense, semantically filtered subgraph that enables partial model loading during inference.

B. Query-Aware Model Instantiation

The KG-WISE query-aware inference pipeline, as illustrated in Figure 4, begins when a user submits an inference query

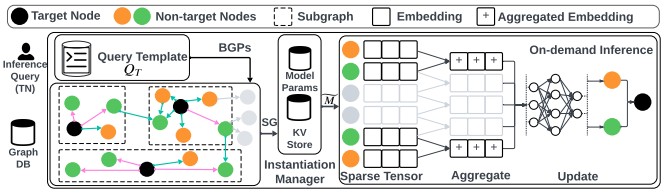

Fig. 4. KG-WISE inference pipeline. Given an inference query, KG-WISE loads a stored SPARQL template to extract a semantically relevant subgraph SG, then instantiates a compact model $\widetilde{M}$ by loading only the required embeddings and weights from the KV store. Inference is executed on-demand using sparse tensor aggregation over SG, avoiding full model loading.

Algorithm 3 Inference Subgraph Extraction

Input: Query Template Q_T, Knoweldge Graph KG, Target Nodes TN, SPARQL Endpoint SP
Output: Inference Subgraph SG
1: **function** Inference_SG_Extractor(Q_T, TN, KG, SP)
2: $Q \leftarrow$ instantiateQuery(Q_T, TN)
3: $Triples \leftarrow$ executeSPARQL(KG, SP, Q)
4: $SG' \leftarrow$ dropDuplicates($Triples$)
5: $SG' \leftarrow$ nodeEncodings(SG', KG)
6: $SG' \leftarrow$ relationEncodings(SG', KG)
7: $SG \leftarrow$ labelEncodings(SG', KG)
8: **return** SG

$\mathcal{Q}_I$ containing a set of target nodes TN. A corresponding LLM-generated SPARQL query is retrieved and used to extract the relevant inference subgraph SG. In Figure 4, gray nodes represent nodes excluded by the query, illustrating how KG-WISE minimizes resource usage by avoiding full-graph loading during inference. Following the extraction of the inference subgraph, KG-WISE receives the encoded subgraph SG and instantiates a query-specific model $\widetilde{M}$ in two steps:

- Step 1 (Model Initialization): The KG-WISE Model Loader initializes the GNN architecture and loads the convolutional weights and biases from the Parameter Store. It builds a mapping from node types and IDs in SG to their corresponding embedding locations in the KV store.

- Step 2 (Sparse Tensor Construction): Instead of loading the full embedding matrix, KG-WISE materializes embeddings only for the nodes that appear in SG. These are inserted into a sparse tensor whose indices correspond to the global node encoding space, while all other entries remain empty. This produces a tensor where only (i, j) positions associated with relevant nodes store values, and all non-relevant nodes are implicitly omitted. The resulting sparse tensor becomes the embedding input for $\widetilde{M}$, allowing the forward pass to operate only over nodes in SG rather than the entire KG.

In Algorithm 4, Lines 2 and 3 initialize the model in memory with the same convolutional layer architecture as that of the model M, then load the trained weights and biases. Lines 4 to 8 construct a dictionary where node types are the keys and their corresponding IDs are the values. Lines 9–14 loop through the constructed dictionary, fetching the respective node embeddings from the KV store for each node type. Line 13 generates a sparse tensor with dimensions matching those of the full model M, containing only the embeddings of the nodes present in the subgraph. Dense tensors are replaced with sparse tensors to construct the on-demand model $\widetilde{M}$. The

Algorithm 4 KG-WISE MODEL INSTANTIATION

Input: *Inference Subgraph (SG); Model File-Store (MFS); Model Definition M_D; Key-Value Store (KVS)*
Output: On-demand Model $\widetilde{M}$.

function CONSTRUCT(G_{IR}, M_D, KVS)
  $\widetilde{M} \leftarrow Init(M_D)$ ▷ Initialize model structure based on M_D
  $\widetilde{M}.loadConvLayers(MFS)$ ▷ Load convolutional layers from MFS
  $\mathcal{V} \leftarrow$ getVertices(G_{IR}) ▷ Get vertices in G_{IR}
  $VT_{dic} = \{\}$ ▷ dictionary of neighbor node types
  for each vertex $v \in \mathcal{V}$ **do**
    $v_{type} \leftarrow$ Type(v) ▷ Determine the type of vertex v
    $VT_{dic}[v_{type}].append(v)$ ▷ Append v to the v_{type} list
  for each pair $(V_{Type}, V_{Idx}) \in VT_{dic}$ **do**
    $KVS_{VT} \leftarrow KVS.load(V_{Type})$ ▷ Load VT chunks
    $nrows, ncols \leftarrow KVS_{VT}.shape()$ ▷ get rows and cols count
    $VT_{emb} \leftarrow KVS_{VT}.getEmb(V_{Idx})$ ▷ get embeddings of V_{Idx}
    $\widetilde{M}_{VT} \leftarrow$ SparseTensor($nrows, ncols, VT_{emb}$) ▷ Create a sparse tensor of VT embeddings with dimensions nrows and ncols
  return $\widetilde{M}$ ▷ Return the constructed model

complexity of instantiating $\widetilde{M}$ depends on the number of query nodes TN and the size of SG.

C. The Prediction Manager

The Prediction Manager is the final module in the inference pipeline, responsible for executing predictions using the $\widetilde{M}$ and the subgraph SG. It dynamically initializes the embeddings for the target nodes (TN) while partially loading non-target node embeddings from $\widetilde{M}$. Once initialized, the Prediction Manager processes the subgraph and embeddings through the GNN model's forward pass, which involves aggregation and update operations tailored to the sparse subgraph. Leveraging optimized sparse tensor operations and memory-efficient representations, this process achieves faster computations and up to 60% reduced memory usage compared to state-of-the-art (SOTA) methods. Equation 3 presents the optimized KG-WISE aggregations for the target nodes TN present in the subgraph SG:

$$h_{SG,i}^{(l+1)} = \sigma\left(\sum_{r \in R_{SG}} \sum_{j \in N_{SG,i}^{r}} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)}\right) \quad (3)$$

In this equation, $h_{SG,i}$ is the feature vector for node i in the query subgraph SG, while R_{SG} and N_{SG} represent the subgraph's relations and nodes. Since $|R_{SG}| < |R|$ and $|N_{SG}| \ll |N|$, the message-passing and aggregation steps are smaller and more efficient than in traditional RGCN methods (Equation 1). Consider the set H of node embeddings defined by Equation 3. As the KG grows, loading H becomes increasingly resource-intensive. To mitigate this, KG-WISE generates H_{SG}, a sparse tensor containing embeddings only for the nodes within the inference subgraph SG. Since $|H_{SG}| \ll |H|$, this significantly reduces memory consumption and accelerates data loading. By integrating query-aware inference with a key-value store and RDF engine, KG-WISE constructs an efficient inference pipeline that minimizes memory usage and inference time while preserving model accuracy.

VI. EXPERIMENTAL EVALUATION

This section presents comprehensive experiments to assess our inference system against SOTA GNN inference methods.

TABLE II
KGS STATISTICS AND GNN TASKS: TASK TYPES (TT) INCLUDE NODE CLASSIFICATION (NC) AND LINK PREDICTION (LP).

TT	Name	KG	#nodes	#edges	#n-type	#e-type	Metric
NC	PV	MAG-42M	42.4M	166M	58	62	ACC
NC	PC	YAGO4	30.7M	400M	104	98	ACC
NC	PV	DBLP-15M	15.6M	252M	42	48	ACC
LP	AA	DBLP-15M	15.6M	252M	42	48	H@10
LP	PO	ogbl-wikikg2	2.5M	17M	9.3K	535	H@10
LP	CA	YAGO3-10	123K	1.1M	23	37	H@10

A. Evaluation Setup

Benchmark Datasets and GNN Tasks: We utilize the KGBen benchmark [14], a large and challenging benchmark with KGs of up to 42 million nodes and 166 million edges. We also used small KGs as some methods do not scale to the large ones. We employ several node classification (NC) and link prediction (LP) tasks, each containing from 100 to 1600 target nodes, to simulate real large-scale workloads. In total, we use six real-world KGs from diverse application domains, such as DBLP [15], MAG [16], and YAGO4 [17]. Given the high computational demands of existing GNN methods for link prediction tasks on large KGs, we incorporate smaller datasets like YAGO3-10 [32], a more compact version of YAGO3, and ogbl-wikikg2 [18], a dataset derived from Wikidata.

Our evaluation encompasses both NC and LP tasks. For NC tasks, we focus on single-label classifications, following established heterogeneous graph datasets from previous studies [33], [34], [12]. We use accuracy as the primary metric to evaluate performance on these NC tasks. We choose the Hits@10 metric to evaluate LP performance following SOTA methods [20], [33], [12]. The train–validation–test splits are derived either from three KG versions across timesteps or from random splits. For each dataset, we follow the KGBen benchmark settings [14]. Table II provides statistics on the KG datasets , the node classification (NC) and link prediction (LP) tasks we used.

The GNN Baseline Inference Methods: We evaluate KG-WISE on node classification (NC) tasks using RGCN-based GNNs with training accelerators like GraphSAINT [19] and IBMB [13], and compare it against adapted inference accelerators including Graph Channel Pruning (GCNP) [8], Degree-Quant (DQ) [9], and GKD [10]. Since the original implementations of GCNP, DQ, and GKD support only homogeneous graphs, we extended them to work with RGCN for heterogeneous KGs. Specifically, we replaced linear layers with quantized versions in DQ, applied lasso-based pruning from GCNP, and used GKD's distillation loss in RGCN. For link prediction (LP), we compare KG-WISE against DQ (as an inference accelerator) and MorsE [20] (as a training accelerator), both adapted to support RGCN. We use the default tuned training parameters provided by each method's original implementation. Furthermore, we compare KG-WISE with KG-TOSA [14] to analyze the effects of subgraph extraction and inference optimization in Section VI-D.

Computing Infrastructure: All inference experiments were executed on an Ubuntu VM with dual 64-core Intel Xeon

TABLE III

MODEL SIZE BREAKDOWN FOR KG-WISE AND BASELINE ACCELERATORS. IN KG-BASED GNNS, MOST OF THE MODEL SIZE COMES FROM NON-TARGET NODE EMBEDDINGS, WHICH BASELINES ALWAYS LOAD IN FULL (CONSTANT MODEL SIZE ACROSS ALL $|TN|$). IN CONTRAST, KG-WISE LOADS ONLY EMBEDDINGS FOR THE EXTRACTED SUBGRAPH, YIELDING A QUERY-DEPENDENT MODEL SIZE. "OOM" DENOTES OUT-OF-MEMORY AND "N/A" NOT APPLICABLE. FOR DQ, WEIGHTS AND EMBEDDINGS CANNOT BE DECOUPLED DUE TO SHARED QUANTIZATION STATISTICS.

Task	**Model statistics**	**Baseline Accelerators**				**KG-WISE's model with different —TN—**				
		GS	**IBMB**	**GCNP**	**DQ**	**100**	**200**	**400**	**800**	**1600**
DBLP (PV) NC	# Parameters	1.86 B	2 B	1.85 B	1.86 B	430.42 M	430.42 M	430.42 M	430.42 M	430.42 M
	Size in disk	6.9 GB	7.4 GB	6.9 GB	6.9 GB	17.1 MB	34.8 MB	50.5 MB	67 MB	121.1 MB
	Size(Weights)/Size(Model)	0.01	0.01	0.01	-	0.5	0.33	0.2	0.11	0.06
	Size(Embeddings)/Size(Model)	0.99	0.99	0.99	-	0.5	0.66	0.79	0.88	0.93
YAGO4 (PC) NC	# Parameters	3.65 B	B	1.83 B	3.65 B	243.7 M	243.7 M	243.7 M	243.7 M	243.7 M
	Size in disk	13.6 GB	13.9 GB	13.6 GB	13.6 GB	18.4 MB	24.1 MB	31 MB	46.4 MB	81.5 MB
	Size(Weights)/Size(Model)	0.01	0.01	0.01	-	0.95	0.86	0.68	0.45	0.35
	Size(Embeddings)/Size(Model)	0.99	0.99	0.99	-	0.04	0.13	0.31	0.54	0.64
MAG (PV) NC	# Parameters	5.35 B	B	OOM	5.35 B	876.26 M	876.26 M	876.26 M	876.26 M	876.26 M
	Size in disk	19.9 GB	20.3 GB	OOM	19.9 GB	17.1 MB	34.8 MB	50.5 MB	67 MB	121.1 MB
	Size(Weights)/Size(Model)	0.01	0.01	0.01	-	0.19	0.17	0.14	0.09	0.05
	Size(Embeddings)/Size(Model)	0.99	0.99	0.99	-	0.80	0.82	0.85	0.90	0.94
DBLP (AA) LP	# Parameters	N/A	N/A	N/A	OOM	49.79 M	49.79 M	49.79 M	49.79 M	49.79 M
	Size in disk	N/A	N/A	N/A	OOM	1.9 MB	3.7 MB	7.1 MB	14.1 MB	27.3 MB
	Size(Weights)/Size(Model)	-	-	-	-	0.3	0.15	0.07	0.03	0.02
	Size(Embeddings)/Size(Model)	-	-	-	-	0.69	0.84	0.92	0.96	0.97
YAGO3 (CA) LP	# Parameters	N/A	N/A	N/A	1.50 M	294.53 K	294.53 K	294.53 K	294.53 K	294.53 K
	Size in disk	N/A	N/A	N/A	69 MB	3 MB	3.6 MB	4.2 MB	4.7 MB	4.8 MB
	Size(Weights)/Size(Model)	-	-	-	-	0.12	0.10	0.09	0.08	0.08
	Size(Embeddings)/Size(Model)	-	-	-	-	0.87	0.89	0.90	0.91	0.92
WikiKG (PO) LP	# Parameters	N/A	N/A	N/A	OOM	49.79 M	49.79 M	49.79 M	49.79 M	N/A
	Size in disk	N/A	N/A	N/A	OOM	8.8 MB	10.4 MB	12.6 MB	16.6 MB	-
	Size(Weights)/Size(Model)	-	-	-	-	0.21	0.12	0.08	0.04	-
	Size(Embeddings)/Size(Model)	-	-	-	-	0.78	0.87	0.91	0.95	-

2.4 GHz CPUs and 256 GB RAM. Virtuoso 07.20.3229 served as the RDF engine, with one instance per KG. A separate VM was used per task for training and inference, and node/edge embeddings were stored in Zarr-Python [35] (v2.17.1). We evaluated KG-WISE using both proprietary LLMs (Gemini 2.5 Flash, ChatGPT 4o, ChatGPT 5) accessed via public APIs, and open-weight LLMs (GPT-oss-20B, DeepSeek-R1-Distill-Qwen-14B, Qwen3-30B-A3B-Instruct-2507) hosted locally on a Compute Canada 16 GB VRAM vGPU via llama-cpp. A 32 GB Nvidia Volta V100 GPU was used for GPU-based inference evaluation.

KG-WISE Implementation and Settings: KG-WISE[1] is implemented in Python 3.8, using PyTorch 2.1 and PyG 2.5. Model parameters are persisted in a file store, while metadata is maintained in an RDF graph. Node embeddings are Xavier-initialized. GraphSAINT [19] is used for NC tasks and RGCN [21] for LP tasks. The LLM-guided sampler supports both proprietary and open-weight LLMs; Gemini 2.5 Flash was used for the reported experiments. The hop limit K is fixed to 2 across tasks to avoid over-smoothing [36]. SPARQL queries are executed in parallel: the target nodes of each inference query are partitioned into batches and retrieved using multi-threaded SPARQL execution.

Why Full-Model Loading Is Wasteful: Table III shows that, for all baseline accelerators, embeddings consistently account for $\sim 99\%$ of the total model size, while weights contribute only $\sim 1\%$, regardless of $|TN|$. This is significant since baselines always load the full model (e.g., 6.9–19.9 GB in DBLP/-MAG/YAGO4), even when the query touches only a tiny fraction of the graph. In contrast, KG-WISE materializes only the subgraph-specific embeddings: for DBLP NC the model shrinks from 6.9 GB to as little as 17 MB (at $|TN|$=100), and even at $|TN|$=1600 remains just 121 MB. Similar reductions appear in MAG (19.9 GB→17–121 MB) and YAGO4 (13.6 GB→18–81 MB). These observations confirm that most of the footprint in KG-GNNs is query-irrelevant, and that partial loading enables two orders of magnitude smaller model instantiation, which we next quantify in memory and latency.

B. Node Classification

This experiment evaluates the performance of SOTA methods under inference workloads for node classification, as shown in Figure 5. We include two SOTA GNN training methods, GraphSAINT [19] and IBMB [13], and two inference accelerators, GCNP [8] and DQ [9]. We also attempted to run the GKD [10] inference accelerator, but it could not scale to our large graphs and ran out of memory in all our experiments. The inference query targets 1K nodes, distributed across all task classes.

KG-WISE achieves comparable or up to 4% higher accuracy across three tasks. This is attributed to our LLM-guided task-oriented subgraph, which yields a smaller and sparser model. The subgraph's sparsity acts as a form of regularization, improving accuracy by excluding irrelevant nodes and edges. KG-WISE achieved up to a 22× speedup in inference time and a 93% reduction in memory usage on the DBLP-PV

[1]KG-WISE is open-sourced at https://github.com/CoDS-GCS/KG-WISE

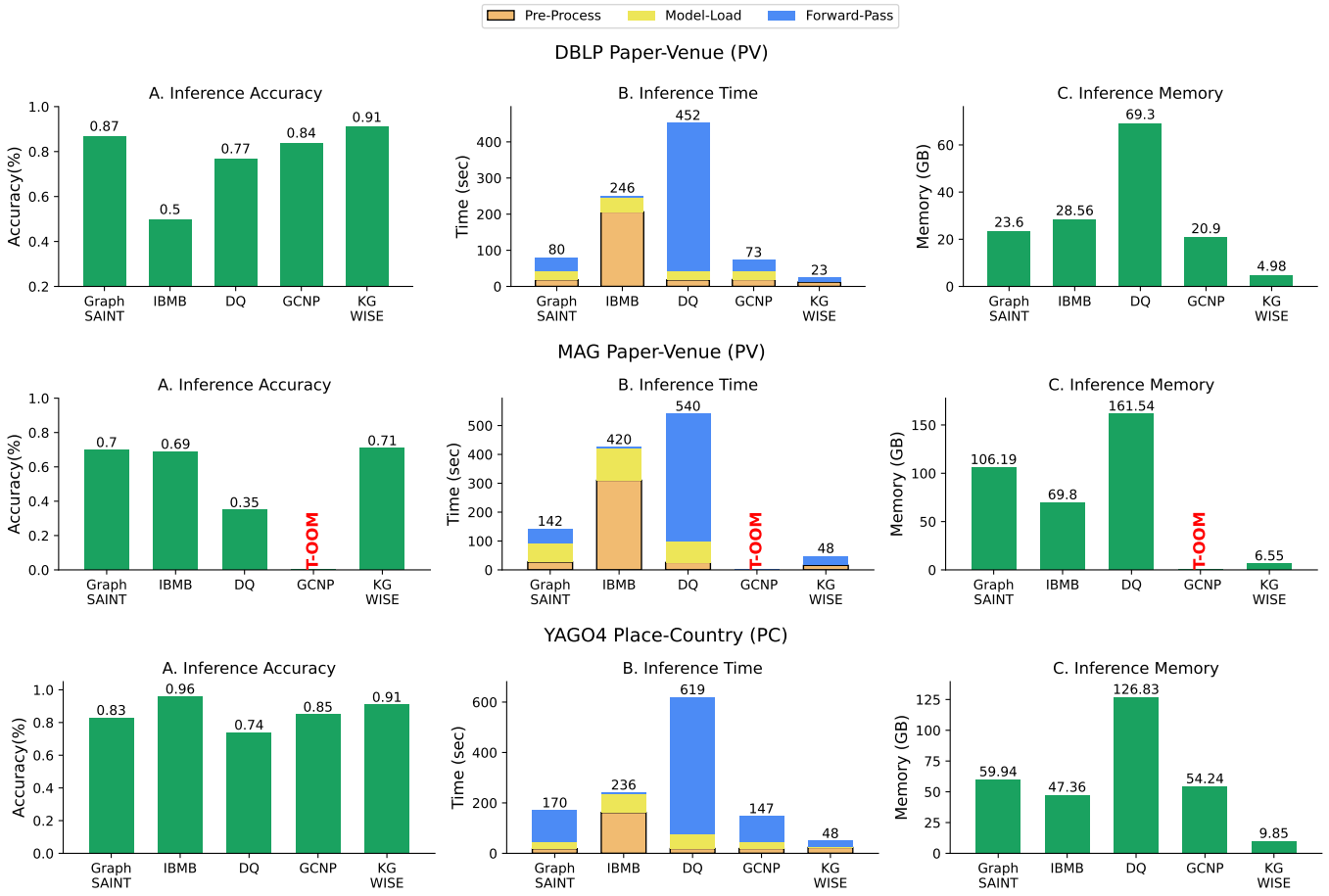

Fig. 5. Performance across NC tasks is based on three metrics: (A) Inference Accuracy (higher is better), (B) Inference-Time (lower is better), and (C) Inference Memory (lower is better). The top and middle sections illustrate the results for the Paper-Venue task on DBLP and MAG, respectively. The bottom figures present the results of the Place-Country task on YAGO4. The inference query performs inference for 1K target nodes stratified across all classes. KG-WISE archives comparable inference accuracy compared with the SOTA training/inference accelerators (Graph SAINT, IBMB, GCNP, and DQ). KG-WISE outperforms the SOTA methods by up to 28x in inference time on YAGO4 with memory reduction up to 98%.

task. On the MAG-PV task, the largest and densest graph, KG-WISE delivered a 10× faster inference and 92% memory savings compared to DQ, while GCNP faced OOM errors. For the YAGO4-PC task, which involves complex node and edge types for place-country classification, KG-WISE achieved a 28× speedup with 98% memory savings over DQ, and a 6× speedup with 93% savings over GCNP.

Training accelerators like GraphSAINT and IBMB achieved the highest accuracy across all tasks but incurred higher sampling overhead and require full model loading into memory. In comparison, KG-WISE maintained competitive accuracy while achieving faster loading, quicker inference, and lower memory footprint. This efficiency stems from the query-specific model constructed during inference, which is substantially smaller, as shown in Table III. Overall, tasks with higher graph sparsity benefited most, aligning with typical characteristics of real-world knowledge graphs [37].

C. Link Prediction

This experiment compares the performance of KG-WISE to two SOTA methods, DQ and MorsE [20], on inference workloads for link prediction on KGs from diverse domains. The results are shown in Figure 6. The inference query consists of 100 target nodes, distributed across different source node types. KG-WISE achieved higher accuracy than DQ and MorsE in all tasks while being significantly faster and more resource-efficient. On the DBLP dataset, the largest of the three KGs in this experiment, KG-WISE outperforms MorsE by achieving 17 points higher Hits@10, 18× faster inference, and 95% lower memory usage. Since link prediction assigns a score to every node in the subgraph, larger graphs increase both noise and computational cost. KG-WISE's subgraph sampler improves accuracy while simultaneously reducing overhead by extracting semantically relevant subgraphs for each query node, thereby focusing the computation on only relevant nodes. A similar pattern is observed in the YAGO3-10-CA dataset, where KG-WISE delivers a 7× speedup and 28% memory reduction over MorsE.

WikiKG is the most heterogeneous KG in our benchmark, with many node and edge types that introduce semantically irrelevant neighbors. KG-WISE's LLM-guided sampler prunes this noise, reducing 535 relation types to just 50. For the Person–Occupation task, it excludes distractors such as input-device or programming-language relations while retaining only informative ones (e.g., digital game entities that reflect user interests). This targeted reduction yields a

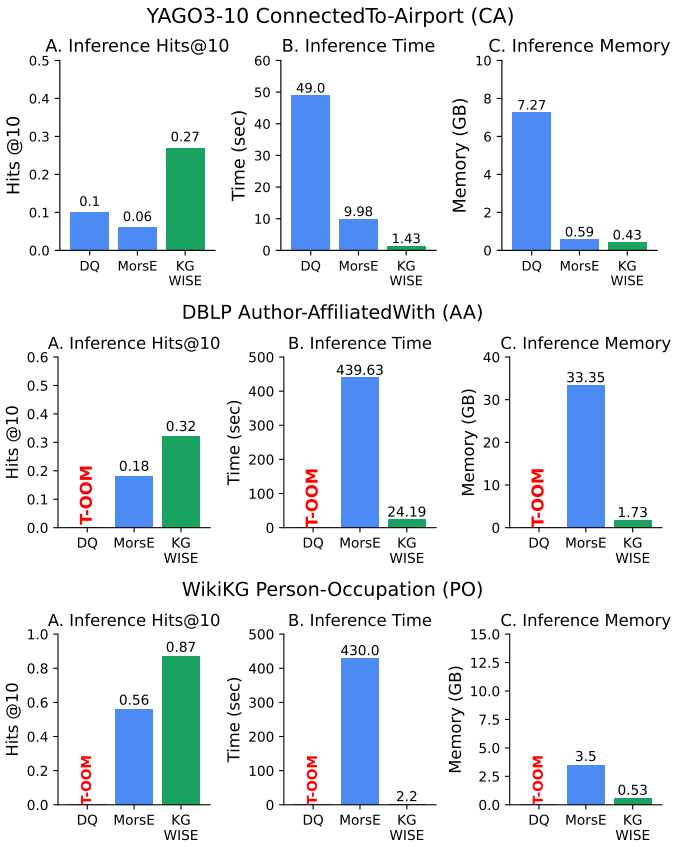

Fig. 6. Results on link prediction tasks across three KGs, reporting (A) Hits@10, (B) inference time, and (C) memory usage. Rows correspond to YAGO3-10 (CA), DBLP (AA), and WikiKG (PO). Each query uses 100 target nodes. KG-WISE improves accuracy while reducing inference time (up to 7×) and memory compared to MorsE and DQ (which fails OOM in two cases).

31-point increase in Hits@10, a 195× speedup, and an 85% memory savings over MorsE.

D. Ablation Study

KG-WISE Scalability: We demonstrate the scalability of KG-WISE through weak and strong scalability tests as shown in Figure 7. In the weak scalability setting, using the DBLP inference dataset, both the problem size (number of queries) and the number of workers (CPU cores) increase proportionally. Each worker operates independently with its own model copy and processes the entire inference pipeline. KG-WISE achieves a time efficiency of 0.68 compared to 0.76 for Graph-SAINT, with better memory efficiency at 0.26 versus 0.24.

In the strong scalability setting, a constant problem size consisting of inference queries from NC tasks namely [DBLP (#200), YAGO (#200), MAG (#200), DBLP (#400), YAGO (#400), MAG (#400), DBLP (#800), YAGO (#800), MAG (#800)] is tested while varying the number of workers (1 to 4, with 4 cores per worker). At 4 workers, KG-WISE achieves a speedup of 2.4 compared to 1.83 for Graph-SAINT, with memory efficiency of 0.25 compared to 0.49. In summary, KG-WISE demonstrates better memory savings in memory-bound settings (weak scalability) and higher computational speed in CPU-bound settings (strong scalability).

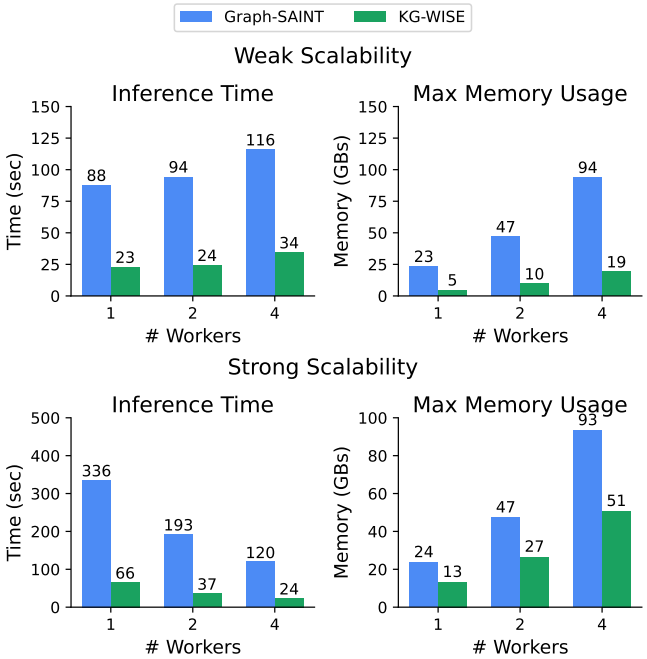

Fig. 7. The inference time and memory consumption of KG-WISE and Graph-SAINT inference pipelines under weak and strong scalability settings. KG-WISE shows better scalability in both settings.

KG-WISE under varying inference-query size. KG-WISE is most effective when the inference query contains a limited number of target nodes. Figures 1 and 2 in the supplementary material report results for node classification and link prediction as $|TN|$ increases. For baselines, both inference time and memory remain almost constant because they always load the full model and graph. In contrast, KG-WISE grows only slightly with $|TN|$, since it loads only embeddings for the extracted subgraph. For example, on DBLP, inference time rises from 25 s to 30 s when $|TN|$ increases from 100 to 1.6K, while the fastest baseline (GCNP) stays around 150 s. Memory usage shows a similar trend: at 1.6K target nodes, KG-WISE uses 6.3 GB versus 30 GB for GCNP (79% reduction). On denser KGs such as MAG and YAGO, the ratio of embedding size to weights is lower (Table III), which narrows the margin, but KG-WISE still outperforms all baselines in both time and memory. For link prediction, the inference time and memory consumption of other methods are constant or increase linearly with the number of target nodes. In contrast, KG-WISE shows sub-linear scaling in both inference time and memory usage.

KG-WISE on GPU. We evaluate the performance of KG-WISE and baseline during inference on a CPU versus a GPU. KG-WISE effectively utilizes GPU hardware to accelerate inference. Figure 8 compares CPU and GPU inference. KG-WISE benefits directly from GPU acceleration: on DBLP, inference time drops from 21 s (CPU) to 13 s (GPU); on MAG from 49 s to 22 s; and on YAGO4 from 34 s to 15 s. In contrast, GraphSAINT (GSAINT) remains slower even when moved to GPU, and in several LP/NC tasks the baseline models run out of GPU memory. KG-WISE avoids these failures by loading only subgraph-specific embeddings, which keeps its GPU footprint small enough to fit within VRAM. GPU memory

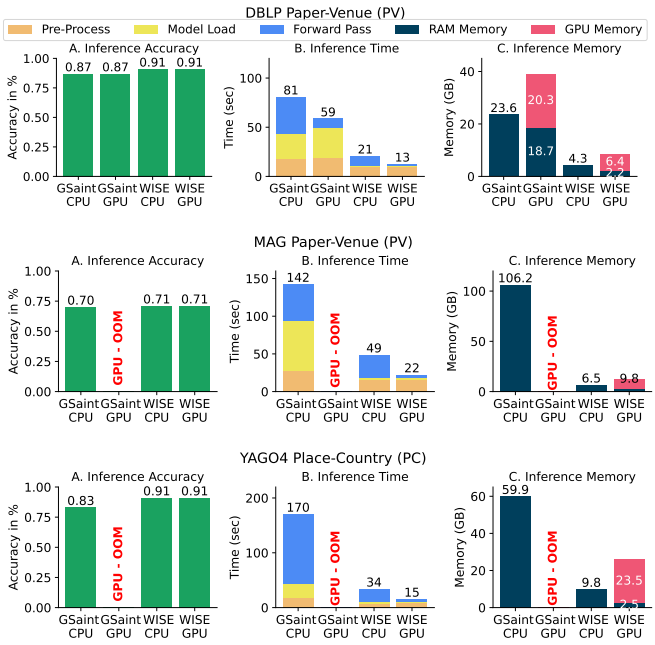

Fig. 8. Evaluation of KG-WISE while running on a CPU and a GPU

usage is higher than CPU RAM usage, as expected, but remains far below the baselines and does not limit execution. These results show that KG-WISE can leverage GPU hardware effectively while retaining its partial-loading advantage.

LLM-Guided Subgraphs for Training and Inference: We compare KG-WISE with KG-TOSA to quantify the effect of LLM-guided subgraph extraction. Both systems use subgraphs during training, but KG-TOSA relies on fixed graph patterns, while KG-WISE selects semantically relevant relations using an LLM-guided query template. We evaluate both approaches across three NC tasks on DBLP, MAG, and YAGO4, using GraphSAINT as the backbone in all cases (Figure 9).

Training impact: Fixed graph patterns in KG-TOSA include many irrelevant triples, which increases subgraph size and training time. In contrast, KG-WISE filters the schema using semantic cues from the LLM, producing more compact training subgraphs. As a result, training time is reduced by 2–3× (e.g., DBLP: 26.9→11.9 min, MAG: 59.3→34.3 min, YAGO4: 57.9→16.8 min). Training memory usage is also up to 78% lower, with accuracy remaining comparable.

Inference impact: During inference, KG-TOSA loads the full model into memory and performs a complete forward pass, while KG-WISE instantiates a compact model by loading only the embeddings and weights required for the extracted subgraph. As shown in Figure 9, this leads to 41–52% faster inference and 60–84% lower memory usage across all datasets. The gains are largest on dense or highly heterogeneous graphs such as MAG and YAGO4, where KG-TOSA loads many irrelevant node embeddings. In contrast, KG-WISE restricts computation to the semantically relevant neighborhood of the query. In the few cases where KG-TOSA achieves slightly higher accuracy, this comes from training on larger subgraphs, but it also causes up to 6× higher memory usage and nearly double the inference time. These results confirm that LLM-guided subgraph selection enables scalable, query-aware inference while preserving accuracy.

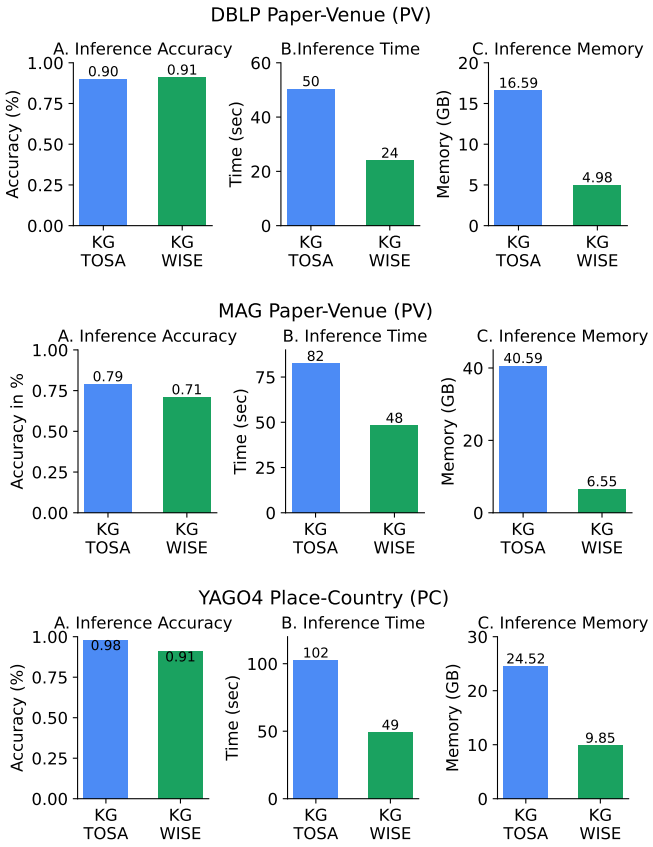

Fig. 9. KG-TOSA vs. KG-WISE on node classification tasks using three metrics: (A) Inference Accuracy (higher is better), (B) Inference Time (lower is better), and (C) Inference Memory (lower is better). Our LLM-guided subgraph extraction and query-aware inference substantially improve efficiency: KG-WISE reduces inference time by 41–52% and memory usage by 60–84% while preserving accuracy.

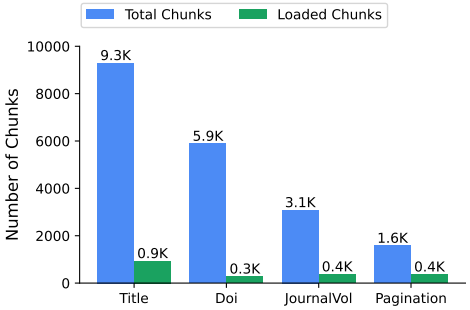

Fig. 10. The number of KV-store chunks loaded per node type by KG-WISE to answer the 1K target node inference query on the DBLP KG. KG-WISE loaded just 10% of the KV-chunks, achieving significant savings in both loading time and memory.

KG-WISE and KV-Chunks Access: The efficient chunking and indexing of neighbor node embeddings in KG-WISE enables the system to load only the embeddings of the nodes specified in the inference subgraph. This selective loading significantly reduces memory overhead and inference time. Figure 10 illustrates the number of loaded chunks per node type for the DBLP NC task using KG-WISE. The figure highlights the top 4 most frequent node types and shows the total number of chunks and the chunks loaded by KG-WISE,

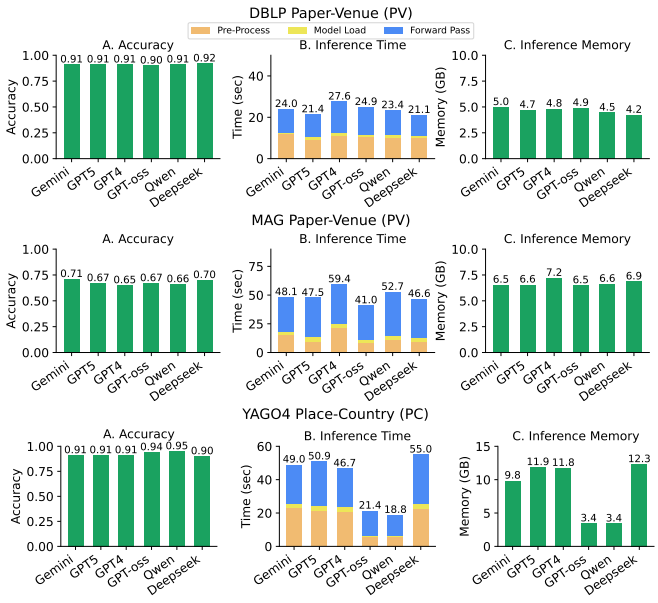

Fig. 11. Evaluation of different LLMs used to generate the query template Q_T in KG-WISE across three KG tasks. Accuracy remains stable across all models, while inference time and memory vary due to differences in predicate selection. Open-weight LLMs (Qwen, GPT-oss) produce more compact subgraphs on heterogeneous KGs (e.g., YAGO4), yielding lower memory usage and faster inference compared to proprietary LLMs.

showcasing the effectiveness of KG-WISE in minimizing unnecessary data loading. Notably, only 10% of the total chunks were required, demonstrating substantial computational and memory savings. By efficiently querying the KV-store indexed chunks, KG-WISE achieves significant savings in embedding loading time and memory consumption resulting in a smaller embedding memory footprint in $\widetilde{M}$. Furthermore, indexing node embeddings within the KV-store facilitates scalable storage for large-scale KG embeddings while ensuring a lightweight GNN inference query complexity, making KG-WISE well-suited for high-throughput applications.

LLM-Agnostic Template Generation and Inference Efficiency. KG-WISE's prompt-engineering pipeline splits query generation into verifiable subtasks (Algorithm 1), which stabilizes LLM behavior and prevents drift. This staged verification ensures that even lightweight open-weight models can generate Q_T of comparable quality to proprietary models. As a result, KG-WISE generalizes across LLM types while enabling a tunable trade-off between subgraph compactness and resource usage. Figure 11 compares KG-WISE across six LLMs. Accuracy remains stable on all datasets, confirming that inference quality does not depend on which LLM produces the template Q_T. The main differences arise in inference time and memory footprint, driven by how each Q_T selects predicates during subgraph extraction. On DBLP, all models behave similarly because the schema is shallow and low in heterogeneity. On MAG, the denser relation structure slightly reduces accuracy across all LLMs but runtime differences remain small. The largest contrast appears on YAGO4: open-weight models (Qwen, GPT-oss) select more compact predicate sets, yielding smaller subgraphs and reducing memory to

TABLE IV
CO_2 EMISSIONS AND ENERGY CONSUMPTION OF KG-WISE AND GRAPH-SAINT. KG-WISE GENERATES 60% LESS CO_2 EMISSIONS AND CONSUMES 62% LESS ENERGY.

Metric	GraphSAINT	KG-WISE
Total Energy (Wh)	1.19	0.42
Energy per Target Node (Wh)	1.2×10^{-3}	0.4×10^{-3}
Total Emission (g CO_2)	1.5×10^{-1}	0.53×10^{-1}
Emission per Target Node (g CO_2)	1.5×10^{-4}	0.53×10^{-4}
Relative CO_2 Efficiency	-	2.83x

$\sim$3.4 GB, whereas proprietary models (Gemini, GPT-4, GPT-5) expand the neighborhood scope and push memory usage to 9–12 GB. These results show that "larger LLM" does not imply "better Q_T." Compact predicate selection often leads to faster and cheaper inference without sacrificing accuracy.

E. Energy and Carbon Emissions.

We measure the inference carbon footprint of KG-WISE and GraphSAINT using CodeCarbon [38]. CO_2 emissions are computed as $C \times E$, where E is the measured energy consumption (CPU + memory) and C is the carbon intensity of the electricity source recorded by CodeCarbon. Only CPU and memory energy are included; memory is estimated at 0.375 W/GB and CPU usage is read from Intel RAPL. Table IV reports the energy usage and CO_2 emissions when running inference on 1K DBLP target nodes. KG-WISE consumes 0.42 Wh compared to 1.19 Wh for GraphSAINT, a 62% reduction. This translates to 3$\times$ lower energy per target node. The total CO_2 emissions drop from 1.5×10^{-1} g to 0.53×10^{-1} g, a 60% reduction. The relative CO_2 efficiency of KG-WISE is therefore 2.83$\times$. These savings stem from avoiding full-model loading and reducing forward-pass computation, showing that KG-WISE improves system efficiency while also lowering environmental cost for better sustainability.

VII. CONCLUSION

In heterogeneous GNNs over large KGs, the majority of the model size comes from embeddings of non-target nodes rather than trainable weights. As our analysis shows, embeddings can account for more than 95% of the total model size, while weights contribute only a small fraction. Existing systems must still load these embeddings in full at inference time, even when only a small task-specific subgraph is relevant. This results in unnecessary memory consumption and computation. KG-WISE addresses this inefficiency by decoupling the GNN model into fine-grained components and enabling query-aware retrieval. An LLM-guided query template, generated once before training, is used to extract a compact task-relevant subgraph for each inference query. KG-WISE then instantiates a smaller model $\widetilde{M}$ by loading only the embeddings and parameters needed for that subgraph, avoiding full model materialization. Our experiments on six real-world KGs show that this design significantly reduces inference memory usage and latency while maintaining or improving accuracy, as well as saving energy. These results demonstrate that partial model loading and query-aware adaptation are practical and effective strategies for scalable and sustainable GNN inference in large heterogeneous KGs.

VIII. AI-Generated Content Acknowledgment

This work utilized GenAI chat models as a supportive tool during manuscript preparation and auxiliary research tasks. The LLM was used to assist with language refinement (e.g., improving clarity and readability of selected sections), to generate candidate subgraph samples under predefined structural constraints for system benchmarking purposes, and to support schema consistency checking against formally defined constraints. All generated content and graph instances were subsequently reviewed, programmatically validated, and verified by the authors. The LLM was not used to produce core scientific contributions, algorithms, experimental results, figures, datasets, or code, and the authors take full responsibility for the integrity and accuracy of the work.

References

[1] Z. Ye, Y. J. Kumar, G. O. Sing, F. Song, and J. Wang, "A comprehensive survey of graph neural networks for knowledge graphs," *IEEE Access*, vol. 10, pp. 75 729–75 741, 2022. [Online]. Available: https://doi.org/10.1109/ACCESS.2022.3191784

[2] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 5, pp. 97:1–97:37, 2023. [Online]. Available: https://doi.org/10.1145/3535101

[3] X. Lin, Z. Quan, Z. Wang, T. Ma, and X. Zeng, "KGNN: Knowledge graph neural network for drug-drug interaction prediction," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2020, pp. 2739–2745. [Online]. Available: https://doi.org/10.24963/ijcai.2020/380

[4] O. A. Ekle and W. Eberle, "Anomaly detection in dynamic graphs: A comprehensive survey," *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 8, pp. 192:1–192:44, 2024. [Online]. Available: https://doi.org/10.1145/3669906

[5] Y. Dou, Z. Liu, L. Sun, Y. Deng, H. Peng, and P. S. Yu, "Enhancing graph neural network-based fraud detectors against camouflaged fraudsters," in *CIKM*, 2020, pp. 315–324. [Online]. Available: https://doi.org/10.1145/3340531.3411903

[6] H. Hu, F. Liu, Q. Pei, Y. Yuan, Z. Xu, and L. Wang, "λgrapher: A resource-efficient serverless system for GNN serving through graph sharing," in *WWW*. ACM, 2024, pp. 2826–2835. [Online]. Available: https://doi.org/10.1145/3589334.3645383

[7] S. Chen and J. Liu, "A survey on graph neural network acceleration: A hardware perspective," *Chinese Journal of Electronics*, vol. 33, no. 3, pp. 601–622, 2024.

[8] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. K. Prasanna, "Accelerating large scale real-time GNN inference using channel pruning," *Proc. VLDB Endow.*, vol. 14, no. 9, pp. 1597–1605, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1597-zhou.pdf

[9] S. A. Tailor, J. Fernández-Marqués, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," in *ICLR*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=NSBrFgJAHg

[10] C. Yang, Q. Wu, and J. Yan, "Geometric knowledge distillation: Topology compression for graph neural networks," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/c06f788963f0ce069f5b2dbf83fe7822-Abstract-Conference.html

[11] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, 2017, pp. 1024–1034. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[12] H. Zeng, M. Zhang, Y. Xia, A. Srivastava, A. Malevich, R. Kannan, V. K. Prasanna, L. Jin, and R. Chen, "Decoupling the depth and scope of graph neural networks," in *NeurIPS*, 2021, pp. 19 665–19 679, , GitHub Code: https://github.com/facebookresearch/shaDow_GNN. [Online]. Available: https://arxiv.org/abs/2201.07858

[13] J. Gasteiger, C. Qian, and S. Günnemann, "Influence-based mini-batching for graph neural networks," in *LoG*, ser. Proceedings of Machine Learning Research, vol. 198, 2022, p. 9. [Online]. Available: https://proceedings.mlr.press/v198/gasteiger22a.html

[14] H. Abdallah, W. Afandi, P. Kalnis, and E. Mansour, "Task-oriented gnns training on large knowledge graphs for accurate and efficient modeling," in *ICDE*, 2024, pp. 1833–1846. [Online]. Available: https://doi.org/10.1109/ICDE60146.2024.00148

[15] M. R. Ackermann. (2022) dblp in rdf. [Online]. Available: https://blog.dblp.org/2022/03/02/dblp-in-rdf/

[16] M. Färber, "The microsoft academic knowledge graph: A linked data source with 8 billion triples of scholarly data," in *ISWC*, ser. Lecture Notes in Computer Science, vol. 11779, 2019, pp. 113–129. [Online]. Available: https://doi.org/10.1007/978-3-030-30796-7_8

[17] T. P. Tanon, G. Weikum, and F. M. Suchanek, "YAGO 4: A reason-able knowledge base," in *ESWC*, ser. Lecture Notes in Computer Science, vol. 12123. Springer, 2020, pp. 583–596. [Online]. Available: https://doi.org/10.1007/978-3-030-49461-2_34

[18] D. Vrandecic and M. Krötzsch, "Wikidata: a free collaborative knowledge base," *Commun. ACM*, vol. 57, no. 10, pp. 78–85, 2014. [Online]. Available: https://doi.org/10.1145/2629489

[19] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "Graphsaint: Graph sampling based inductive learning method," in *ICLR*, 2020, , GitHub Code: https://github.com/snap-stanford/ogb/blob/master/examples/nodeproppred/mag/graph_saint.py.

[20] M. Chen, W. Zhang, Y. Zhu, H. Zhou, Z. Yuan, C. Xu, and H. Chen, "Meta-knowledge transfer for inductive knowledge graph embedding," in *ACM SIGIR*, ser. SIGIR '22, 2022, p. 927–937, , GitHub Code: https://github.com/zjukg/MorsE. [Online]. Available: https://doi.org/10.1145/3477495.3531757

[21] M. S. Schlichtkrull, T. N. Kipf, and e. a. Peter Bloem, "Modeling relational data with graph convolutional networks," in *ESWC*, vol. 10843. Springer, 2018, pp. 593–607, , GitHub Code: https://github.com/thiviyanT/torch-rgcn. [Online]. Available: https://doi.org/10.1007/978-3-319-93417-4_38

[22] D. Blakely, J. Lanchantin, and Y. Qi, "Time and space complexity of graph convolutional networks," vol. 31, p. 2021, 2021. [Online]. Available: https://qdata.github.io/deep2Read/talks-mb2019/Derrick_201906_GCN_complexityAnalysis-writeup.pdf

[23] H. You, Z. Lu, Z. Zhou, Y. Fu, and Y. Lin, "Early-bird gcns: Graph-network co-optimization towards more efficient GCN training and inference via drawing early-bird lottery tickets," in *AAAI*. AAAI Press, 2022, pp. 8910–8918. [Online]. Available: https://doi.org/10.1609/aaai.v36i8.20873

[24] B. Feng, Y. Wang, X. Li, S. Yang, X. Peng, and Y. Ding, "Sgquant: Squeezing the last bit on graph neural networks with specialized quantization," in *ICTAI*. IEEE, 2020, pp. 1044–1052. [Online]. Available: https://doi.org/10.1109/ICTAI50040.2020.00198

[25] B. Yan, C. Wang, G. Guo, and Y. Lou, "Tinygnn: Learning efficient graph neural networks," ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1848–1856. [Online]. Available: https://doi.org/10.1145/3394486.3403236

[26] S. Zhang, Y. Liu, Y. Sun, and N. Shah, "Graph-less neural networks: Teaching old mlps new tricks via distillation," in *ICLR*. OpenReview.net, 2022. [Online]. Available: https://openreview.net/forum?id=4p6_5HBWPCw

[27] T. Liu, P. Li, Z. Su, and M. Dong, "Efficient inference of graph neural networks using local sensitive hash," *IEEE Trans. Sustain. Comput.*, vol. 9, no. 3, pp. 548–558, 2024. [Online]. Available: https://doi.org/10.1109/TSUSC.2024.3351282

[28] W. Afandi, H. Abdallah, A. Aboulnaga, and E. Mansour, "An llm-guided query-aware inference system for gnn models on large knowledge graphs," *arXiv preprint arXiv:2603.04545*, 2026. [Online]. Available: https://arxiv.org/abs/2603.04545

[29] J. Moore and S. Kunis, "Zarr: A cloud-optimized storage for interactive access of large arrays," in *Proceedings of the Conference on Data Infrastructure*, vol. 1, 2023.

[30] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," 2024.

[31] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *International Conference on Machine Learning*, 2020. [Online]. Available: https://arxiv.org/abs/1908.10396

[32] T. Dettmers, P. Minervini, P. Stenetorp, and S. Riedel, "Convolutional 2d knowledge graph embeddings," in *AAAI*, 2018, pp. 1811–1818. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17366

[33] W. Hu, M. Fey, M. Zitnik, and Y. D. et.al., "Open graph benchmark: Datasets for machine learning on graphs," in *NeurIPS*, 2020.

[34] X. Yang, M. Yan, S. Pan, X. Ye, and D. Fan, "Simple and efficient heterogeneous graph neural network," *AAAI*, vol. abs/2207.02547, 2023, gitHub Code: https://github.com/ICT-GIMLab/SeHGNN. [Online]. Available: https://doi.org/10.48550/arXiv.2207.02547

[35] D. Stansby. (2024) zarr-python. [Online]. Available: https://github.com/zarr-developers/zarr-python

[36] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence IAAI 2020*. AAAI Press, 2020, pp. 3438–3445.

[37] S. Rezayi, H. Zhao, and et al., "Edge: Enriching knowledge graph embeddings with external text," in *NAACL-HLT*, 2021, pp. 2767–2776. [Online]. Available: https://doi.org/10.18653/v1/2021.naacl-main.221

[38] B. Courty, V. Schmidt, S. Luccioni, and et.al., "mlco2/codecarbon: v2.4.1," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11171501