

APEX: Adaptive Variable-wise Parallel Execution for Worst-Case Optimal Joins on Graph Queries

Yipeng Liu[†], Yuming Lin^{*†}, Zhicheng Pan[‡], Chengcheng Yang^{*‡}, You Li[†], Aoying Zhou[‡]

[†]Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, China

23032303001@mails.guet.edu.cn, {ymlin, liyou}@guet.edu.cn

[‡]School of Data Science and Engineering, East China Normal University, Shanghai, China

zcpa@stu.ecnu.edu.cn, {ccyang, ayzhou}@dase.ecnu.edu.cn

Abstract—Graph queries are ubiquitous in graph databases and knowledge graphs, supporting applications such as recommender systems, question answering, and semantic search. In recent years, worst-case optimal join (WCOJ) algorithms play a critical role in evaluating graph queries. However, their practical performance is hindered by load imbalance in parallelization and poor execution order of join variables. In this paper, we propose APEX, Adaptive variable-wise Parallel EXecution for WCOJ on graph queries. First, APEX parallelizes the core intersection operations in Leapfrog Triejoin by materializing and organizing all intersection results for a variable at once. This is enabled by a candidate propagation graph, which captures dependencies among variables and their intersection results. Second, we further propose a reinforcement learning policy with a graph convolutional network encoder that selects the next variable based on query structure and runtime feedback. To accelerate lookups on the candidate propagation graph, we introduce two techniques: (i) *dependency-source detection* to prune the traversal space, and (ii) a *Bloom filter* to avoid redundant checks. Extensive experiments on diverse real-world and synthetic datasets demonstrate that APEX achieves up to 2.42–5.79 $\times$ speedup over state-of-the-art baselines.

Index Terms—worst-case optimal join, query optimization, join ordering, graph queries

I. INTRODUCTION

From social networks to knowledge graphs, modern applications increasingly rely on complex graph patterns that demand multi-way joins over massive graphs. However, traditional query processors evaluate such patterns using the binary join paradigm [1], [2], which decomposes multi-way joins into sequences of binary join operations. While effective for *acyclic* queries, this approach suffers from severe intermediate-result explosion on *cyclic* workloads: the size of intermediate results can grow exponentially in the input size, far exceeding the final output cardinality [3]. Consequently, they fail to scale for many common cyclic patterns in real-world graph analytics.

This limitation has spurred significant interest in worst-case optimal join (WCOJ) algorithms [4], which guarantee query runtimes proportional to the worst-case output size. Among WCOJ methods, Leapfrog Triejoin (LFTJ) [5]–[8] has emerged as a particularly influential approach. LFTJ evaluates multi-way joins via ordered variable-at-a-time traversal over trie-structured data, leveraging intersection-based probing to

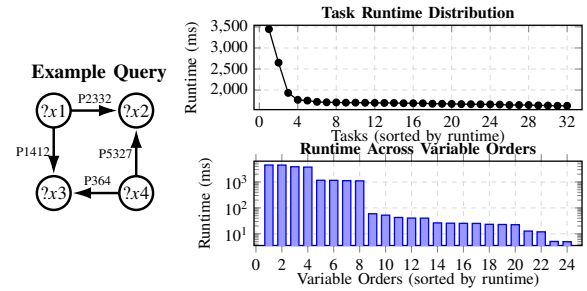


Fig. 1. **Left:** Example query from the WGPB [13]. **Right Top:** Task runtime distribution based on first variable's domain partitioning, showing severe load imbalance. **Right Bottom:** Performance (log scale) of different variable orders, with the x-axis sorted by runtime.

prune the search space early and avoid materializing large intermediates. Owing to its strong theoretical guarantees and practical efficiency, WCOJ has been widely adopted as a fundamental building block in modern systems, including graph databases [9], [10], relational engines [11] and so on [12].

However, to maximize the efficiency of WCOJ, the following three challenges need to be addressed.

Efficient Parallelization Implementation (C1). Although WCOJ algorithms offer strong theoretical guarantees, the efficient parallelization implementation of them remains challenging. Specifically, they assign values to variables in a Depth-First Search (DFS) manner, alternating between traversal and intersection, which makes it difficult to parallelize costly intersection operations. Most parallel WCOJ systems only partition the domain of the first variable [11], [12], [14]. The domain is split into fragments, each assigned to a thread; remaining variables are processed sequentially. Such an approach often causes load imbalance, especially when downstream intersections vary or data is skewed. For example, we illustrate this phenomenon with a query from WGPB [13] in Fig. 1 (left). This query is evaluated with variable order $?x4 \ ?x3 \ ?x1 \ ?x2$, using parallelization by partitioning the first variable's domain. Fig. 1 (top-right) shows runtime per fragment: a few fragments dominate, most finish quickly. This highlights the limitation of partitioning only the domain of the first variable, resulting in poor parallel utilization.

*Corresponding author

Dynamic Variable Processing (C2). Existing WCOJ systems [10], [15], [16] primarily rely on static variable orders determined before execution. They strictly follow these orders at runtime, which can lead to poor execution efficiency if the chosen order is suboptimal. By contrast, dynamically adjusting the variable order can mitigate this issue to some extent.

However, adjusting the variable order during query execution is challenging. This is because WCOJ explores the search space of each variable sequentially and enumerates results one by one. This strict sequential procedure makes dynamic re-ordering difficult, as adjusting the variable order requires re-identifying the search space already explored by each variable, which leads to substantial redundant computation. Although the existing work [17] leveraged cubes to track unexplored search space, the domains of the data lists retrieved from the indices are not bounded. Then, before the intersection, it must determine the lower bound of the cube within each list to be intersected, which costs additional overhead. Moreover, frequent variable order changes require re-sorting data, while the original LFTJ sorts only once before execution.

High-Quality Variable Ordering (C3). Even if redundant computation during variable reordering in WCOJ can be avoided, determining an effective variable order still remains a challenging problem. The performance of WCOJ is sensitive to the choice of variable order [4]. In Fig. 1 (bottom-right), our evaluation shows that the fastest variable order runs over two orders of magnitude faster than the slowest. Variable order selection is analogous to join ordering in binary joins, and recent research has explored reinforcement learning (RL) to discover optimal orders [18]–[21].

However, existing methods often fail to produce consistently effective variable orders, as they rely on statistics and heuristics [12], [22], [23] that may lead to suboptimal orders. ADOPT is the first to apply RL to WCOJ, dividing query execution into episodes. Within each episode, a fixed variable order is used, which is subsequently refined between episodes through reinforcement learning. However, ADOPT cannot guarantee an effective initial variable order, which may lead to poor performance during the early episodes.

To address C1 and C2, we propose an adaptive variable-wise parallel WCOJ framework. We achieve this by focusing on parallelizing the intersection operations of LFTJ, as it involves numerous costly intersections during execution. Fortunately, we found that LFTJ’s DFS-style execution is the primary bottleneck for parallelism; therefore, we convert LFTJ to a breadth-first (BFS) style execution, which enables more effective parallelization. To enable this, we introduce the *Candidate Propagation Graph* (CPG) to record dependencies between variables and manage the candidate values of processed variables. We traverse the CPG to prepare intersection inputs for the processing variable and distribute intersection tasks across multiple threads.

Besides, we apply two core optimizations to reduce redundancy in CPG traversal (i) identify each variable’s dependency source to prune traversal in the CPG; and (ii) use a lightweight filter mechanism to prune invalid paths on the fly. Since our

implementation of LFTJ follows a BFS-style, all candidate values of a given variable can be obtained simultaneously. This property ensures that the evaluation of subsequent variables does not affect the results of previously processed ones, thereby enabling dynamic variable selection guided by runtime feedback.

Finally, to address C3, we employ an RL-based variable selector, which includes a graph convolutional network (GCN) encoder over the query graph and a policy network to choose the next variable. This policy network can not only learn from already processed variables but also adaptively select the next variable based on the runtime feedback of processed variables.

In summary, we make the following contributions.

- We propose **APEX**, Adaptive variable-wise Parallel **EX**-ecution for worst-case optimal joins on graph queries, which transforms the LFTJ’s DFS-style execution into a BFS-style execution and employs a *Candidate Propagation Graph* to efficiently organize candidate values.
- We design a reinforcement learning-based variable selector with a graph convolutional encoder, which captures query graph structure and leverages runtime feedback to select the next variable effectively.
- We introduce two redundancy-reduction techniques: dependency-source detection to prune unnecessary traversal and Bloom filters to prune invalid paths.
- We conduct extensive experiments on real-world and synthetic datasets, where APEX consistently outperforms state-of-the-art systems by up to 2.42–5.79×.

II. BACKGROUND AND RELATED WORK

A. Graph and Queries

Data Graph. A data graph is *directed* and *edge-labeled* (see Fig. 2). The graph can be treated as a single ternary relation $R(S, P, O)$. Each tuple (s, p, o) in the relation R corresponds to an edge from vertex s (subject) to vertex o (object) labeled by the predicate p .

Basic Graph Pattern. *Basic Graph Patterns* (BGP) are the foundational building blocks of graph query languages such as SPARQL and Cypher [24]. A basic graph pattern can be seen as a query graph (see Fig. 2), which is matched against the data graph and returns all matched subgraphs. The edges in the query graph are represented by *triple patterns*. Specifically, each triple pattern is represented as a tuple (s_t, p_t, o_t) , where each element can be either a constant or a variable. Moreover, s_t and o_t denote the source and target vertices, and the edge connecting them is labeled by the predicate p_t .

Solution. For a BGP query Q , let $\text{vars}(Q)$ denote the set of variables in Q , and let $C(G)$ denote the set of constants (i.e., vertices and edge labels) in a data graph G . A solution to Q over G is a specific mapping from $\text{vars}(Q)$ to $C(G)$. Specifically, after replacing the variables in Q with their corresponding constants from this mapping, the resulting graph must be a subgraph of G .

Evaluation. The evaluation of Q is a process of finding all the solutions. Fig. 2 shows a BGP query evaluated on the data graph, producing two solutions.

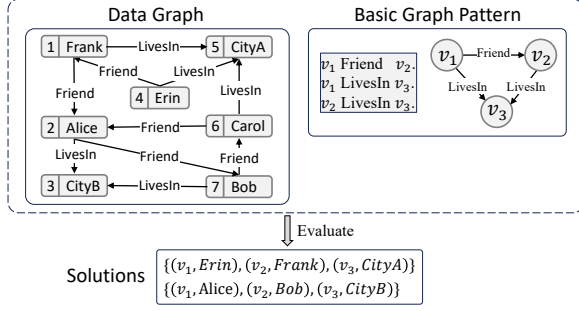


Fig. 2. Data Graph, Basic Graph Pattern, and Solutions.

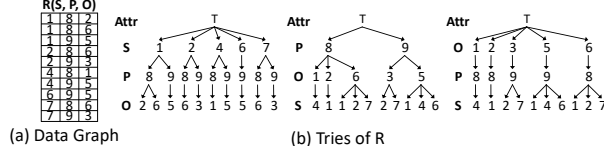


Fig. 3. Data Graph and Trie Example.

B. Worst-case Optimal Join

Worst-case optimal join (WCOJ) algorithms [3], [5] ensure execution time is proportional to the worst-case output size, which is upper-bounded by the Atserias–Grohe–Marx (AGM) bound [3], [25]. The AGM bound is derived from a fractional edge cover of the query hypergraph, representing the maximum number of possible output tuples. The advantage of WCOJ becomes significant for cyclic join patterns, where traditional binary joins often generate large intermediate results.

Leapfrog Triejoin (LFTJ) is a representative WCOJ algorithm that evaluates joins in a variable-at-a-time manner [5]. Unlike binary joins, which process pairs of triple patterns sharing a variable, LFTJ simultaneously processes all triple patterns involving the same variable. Specifically, LFTJ first decides an order in which variables in a query will be processed, called the variable order. Then, following this order, LFTJ assigns a specific concrete value from the data graph to each variable. The operation of assigning a value to a variable is denoted as a *binding*. When all variables have been assigned a binding, a solution of the query is obtained.

More specifically, when processing a variable v , LFTJ first identifies all triple patterns in the query that contain v . For each triple pattern, LFTJ retrieves matching edges from the data graph and, based on the position of v in the pattern (subject, predicate, or object), extracts the corresponding values to form a value list for v that satisfies the pattern. LFTJ then intersects these lists to obtain the candidate values of v , i.e., values that satisfy all triple patterns containing v .

To support efficient value list construction, LFTJ builds multiple trie indexes with different attribute orders (e.g., (O, P, S)), where each trie has three levels and each root-to-leaf path represents one tuple in the specified order. Fig. 3b illustrates the tries for relation R in Fig. 3a, which represents the data graph in Fig. 2, where nodes and edges are mapped to unique IDs based on their labels (e.g., $\text{Friend} \rightarrow 8$ and $\text{LivesIn} \rightarrow 9$). For example, for the pattern $(v, 9, 5)$ with

variable v in the subject position, we use the constant prefix $(9, 5)$ to look up in the (P, O, S) trie and obtain v 's value list, e.g., $[1, 4, 6]$. However, a (P, O, S) trie only supports lookups from predicates to objects or subjects, so LFTJ maintains multiple tries with different attribute orders.

LFTJ applies LEAPFROG JOIN (LFJ) to compute intersections of multiple ascending sorted lists, where each value in the result is a valid binding for v . LFTJ selects one such value, binds it to v , and proceeds to the next variable v' . When processing v' , previously bound variables in its triple patterns are replaced with constants, and the same process of list retrieval and intersection is applied. If the intersection result for v' is empty, LFTJ backtracks to the previous variable and selects the next available binding. If all values in v 's intersection result have already been used as bindings, LFTJ continues backtracking to the variable preceding v . The execution terminates once all possible bindings of the first variable have been explored.

C. Related Work

Parallel WCOJ in Practice. Several systems have explored practical approaches to parallelizing WCOJ. Prior work, such as LogicBlox [12] and Umbra [11], adopts a simple parallelization strategy that partitions the domain of the first variable across threads, with each thread processing the remaining variables sequentially. Although lightweight, this approach often suffers from execution-time imbalance due to significant variation in downstream intersection costs across partitions.

HoneyComb [15] proposes a shared-memory parallelization of WCOJ using Hypercube-style partitioning to distribute all data related to join variables across threads. To eliminate synchronization overhead, it constructs lock-free CoCo indexes. This effectively reduces workload skew and redundant computation. While this design achieves high throughput on multicore CPUs, it requires partitioning the data before execution, and its join order is fixed, limiting flexibility.

Variable Order Selection in WCOJ. APEX leverages Reinforcement Learning (RL) to sequentially determine variable ordering. This idea is similar to the prior work employing RL or other learning-based techniques for database knob tuning [26]–[32] or relational query optimizer [18]–[21]. However, few works have specifically explored the use of RL for *variable order selection* in WCOJ.

Generally, most systems rely on heuristics [22] or statistics [12], [23] to guide the variable order selection. In contrast, ADOPT [17] is the first adaptive WCOJ approach that uses RL to adjust variable orders dynamically during execution. ADOPT divides query execution into episodes, evaluates the query with a fixed order within each episode, and employs the Upper Confidence Bound for Trees (UCT) algorithm to update the variable order between episodes. However, ADOPT only adapts the variable order within the execution of a single query, but it does not learn from evaluated queries. As a result, a poor initial order can degrade the performance of early episodes, since the system has no prior experience to guide the choice of better orders at the start.

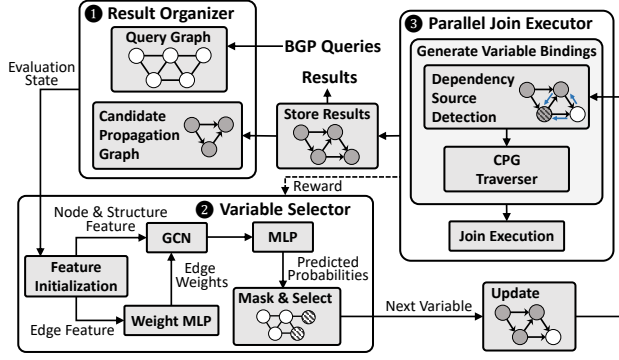


Fig. 4. Overview of APEX.

III. MAIN PROCEDURE OF APEX

A. Overview of APEX

Fig. 4 shows APEX’s architecture. Given a BGP query, APEX repeatedly (i) selects the next variable via the *Variable Selector*, (ii) processes it using the *Parallel Join Executor*, and (iii) organizes the results in the *Result Organizer*, until all variables are handled and final solutions are produced. The three components are summarized below.

❶ **Result Organizer:** Upon receiving a query, APEX constructs a query graph to represent the query and initializes an empty CPG. At each step, a variable is selected, the corresponding vertex and its incident edges are inserted into the CPG according to the query graph. After the variable is processed, the results are organized in this vertex. APEX incrementally adds variables to the CPG until all are handled, then traverses it to produce the final results.

❷ **Variable Selector:** At each step, APEX chooses the next variable based on the query graph and CPG, using learned models. Specifically, APEX extracts node and edge features from the query graph and CPG to represent the current state. Edge weights from an MLP are combined with vertex features and propagated by a two-layer GCN. The policy head outputs selection probabilities, and a mask restricts actions to unprocessed neighbors of processed variables.

❸ **Parallel Join Executor:** Given a selected variable, the executor identifies its dependency source, traverses the CPG in parallel to gather bindings of processed variables, applies filters to reduce redundant accesses, retrieves value lists from the data graph, computes intersections in parallel, and materializes all intersection results in the new CPG vertex.

B. Main Procedure

As discussed in Section II-B, the original LFTJ assigns values to variables in a DFS manner, alternating between variable traversal and intersection computation. This interleaving makes it challenging to parallelize the expensive intersection operations. Transforming LFTJ’s DFS-style evaluation into BFS-style processing allows intersection operations to be performed in batches, enabling more effective parallelization. However, variables in a query may depend on multiple others, resulting in an intricate dependency structure that complicates

Algorithm 1: End-to-end workflow of APEX.

Input: BGP Q , number of threads n
Output: Query result R

```

1 Function APEX( $Q, n$ ):
  // Initialize query graph and CPG
2   $Q = \{V(Q), E(Q)\} \leftarrow \text{QueryGraph}(Q)$ 
3   $\mathcal{G} \leftarrow \emptyset$ 
  // Processed variables
4   $\Phi \leftarrow []$ 
5  while  $|V(Q)| \neq |\Phi|$  do
6     $v' \leftarrow \text{RLVariableSelector}(Q, \mathcal{G})$ 
    // Add vertex and edges
7     $\mathcal{G} \leftarrow \text{UpdateResultGraph}(Q, \mathcal{G}, v')$ 
    // Preparing for joins
8     $B \leftarrow \text{GenerateVariableBinding}(Q, \mathcal{G}, \Phi, v', n)$ 
9     $\text{ParallelJoin}(B, \mathcal{G}, v', n)$ 
10    $\Phi \leftarrow \Phi \parallel [v']$ 
11   $\text{result} \leftarrow \text{CPGTraverse}(\mathcal{G}, \Phi, n)$ 
12  return result

```

BFS traversal. Moreover, to execute intersections in batches, all value lists involved in the intersections must be obtained in advance. To efficiently manage such dependencies and generate the required lists for intersection, we introduce the Candidate Propagation Graph (CPG), denoted as $\mathcal{G}$ (Section IV).

We also construct an undirected query graph Q for a given query Q . Let $V(\cdot)$ and $E(\cdot)$ denote the vertex and edge sets of a graph. Each variable in query Q corresponds to a vertex in $V(Q)$, and each triple pattern involving two variables v_i and v_j is represented by an undirected edge $e = \{v_i, v_j\} \in E(Q)$. The edge is labeled by the fixed element in the triple pattern (i.e., subject, predicate, or object), denoted as $\text{label}(e)$.

For simplicity, we assume all triple patterns have exactly two variables, the common case in practice; single-variable patterns are treated by representing constants as singleton domains. We also assume no two triple patterns share the same variable pair with different constants, making Q a simple graph. A variable is denoted by the same symbol as its corresponding vertex in Q or $\mathcal{G}$ in the following sections.

Note, we refer to the value lists retrieved from the data graph as *intersection lists*. The result of intersecting these lists for a variable is its *candidate list*, which contains all its candidate values that can be regarded as a binding of a variable. A *binding* is denoted as (x, c) , representing the assignment of value c to variable x , and a *binding combination* is a set of bindings that satisfies the corresponding triple patterns.

Main Procedure. The overall procedure of APEX is shown in Algorithm 1. APEX first constructs the query graph Q from the input query Q (Line 2), then initializes the CPG $\mathcal{G}$ and the list of processed variables Φ (Lines 3–4). It then iteratively processes variables in $V(Q)$ through four steps: (i) select the next variable v' based on the current states of Q and $\mathcal{G}$ (Line 6); (ii) insert v' and its incident edges

from Q into $\mathcal{G}$ (Line 7); (iii) traverse $\mathcal{G}$ to generate binding combinations of the previously processed variables, preparing for intersection operations (Algorithm 3, Line 8); and (iv) retrieve the intersection lists and compute the candidate list for v' in parallel using LEAPFROG JOIN which is the intersection operation of LFTJ, storing the results in $\mathcal{G}$ (Line 9). The order of processed variables Φ is then updated by appending v' , where the operator $\parallel$ denotes list concatenation (Line 10). After processing all variables, APEX performs a final traversal over the $\mathcal{G}$ to enumerate final solutions (Lines 11–12).

IV. CANDIDATE PROPAGATION GRAPH

To enable BFS-style execution of LFTJ, all intersections of each variable must be performed at once. As discussed in Section II-B, for a variable v , each triple pattern containing v contributes one intersection list, and the candidate values of v are obtained by intersecting these lists. The triple patterns involving v can be divided into two categories, depending on whether they contain variables that precede v in the variable order and have bindings. In the first category, other variables besides v in the pattern are already bound. Before retrieving the corresponding list, these variables are replaced with their bound constants, and the list is obtained from the updated pattern. In the second category, no variables are bound, so the list is retrieved based on the constants in the pattern.

Since variable bindings evolve during execution, value lists from the first category depend on the current bindings, whereas those from the second category remain fixed. Consequently, the intersection results for v depend on the bindings of variables in the first category. These variables share a triple pattern with v and precede it in the variable order, and we refer to them as the *dependency variables* of v . To compute all intersections for v , it is therefore necessary to enumerate all possible binding combinations of its dependency variables.

Since a query is represented as a query graph Q , where vertices denote variables and edges represent triple patterns. The dependency variables of a variable v are its neighbors in Q that precede it in the variable order. Each neighbor contributes an intersection list: for dependency variables, the list is determined by their bindings and the edge label, while for other neighbors, it is determined only by the edge label. Since the intersection result of a variable is called its candidate list, and each value in this list represents a binding. Specifically, the candidate list of v depends on the bindings of its dependencies.

To record these dependencies and manage the propagation of bindings, we construct a *Candidate Propagation Graph* ($\mathcal{G}$). Each vertex in $V(\mathcal{G})$ corresponds to a processed variable, and each directed edge in $E(\mathcal{G})$ represents a dependency between variables. The CPG is expanded incrementally during evaluation. When a variable v' is selected, a corresponding vertex is added to $V(\mathcal{G})$. For each edge u, v' in Q where u already exists in $\mathcal{G}$, a directed edge $u \rightarrow v'$ is inserted into $E(\mathcal{G})$, indicating that the candidate values of v' depend on u and are propagated from u to v' . For simplicity, we treat vertices representing the same variable in Q and $\mathcal{G}$ as identical.

Algorithm 2: Traversal on $\mathcal{G}$.

Input: Candidate propagation graph $\mathcal{G}$, final result R , variable binding set σ , variable order Φ , index idx , number of threads n , thread id t

```

1 Function Traverse ( $\mathcal{G}, R, \sigma, \Phi, idx, n, t$ ):
2   if  $idx = |\Phi|$  then
3     Insert  $\sigma$  into  $R$ 
4     return
5    $v \leftarrow \Phi[idx]$ 
6   // Bindings of neighbors
7    $b \leftarrow \{(u, \sigma[u]) \mid u \in \text{In}_{\mathcal{G}}(v)\}$ 
8   // Candidate list for  $v$ 
9    $cands \leftarrow \mathcal{G}[v][b]$ 
10  if  $idx = 0$  then
11    // Split domain for parallelism
12     $cands \leftarrow$  the  $t$ -th of  $n$  equal partitions of  $cands$ 
13  for  $value \in cands$  do
14     $\sigma \leftarrow \sigma \cup \{(v, value)\}$ 
15    Traverse ( $\mathcal{G}, R, \sigma, \Phi, idx+1, n, t$ )
16    remove  $(v, value)$  from  $\sigma$ 

```

Input: Candidate propagation graph $\mathcal{G}$, variable order Φ , number of threads n

Output: Solution list R

```

14 Function CPGTraverse ( $\mathcal{G}, \Phi, n$ ):
15    $R \leftarrow []$ 
16   for  $1 \leq t \leq n$  in parallel do
17      $\sigma \leftarrow \emptyset$ 
18     Traverse ( $\mathcal{G}, R, \sigma, \Phi, 0, n, t$ )
19   return  $R$ 

```

Each vertex $v \in V(\mathcal{G})$ takes a combination of bindings from its in-neighbors as input and produces a non-empty candidate list for v as output. Let $\text{In}_{\mathcal{G}}(v)$ denote the set of in-neighbors of v in $\mathcal{G}$, representing the variables on which v depends, and let $\text{Dom}(v)$ denote the domain of possible values that v can take in the data graph. The input of v consists of all possible binding combinations of its in-neighbors:

$$\mathbf{B}_v = \{ \{ (v_i, b_{v_i}) \}_{v_i \in \text{In}_{\mathcal{G}}(v)} \mid b_{v_i} \in \text{Dom}(v_i) \}. \quad (1)$$

To execute all intersection operations for a variable, bindings of previously processed variables must be enumerated. Algorithm 2 traverses the $\mathcal{G}$ using DFS following the variable order Φ , mirroring the traversal strategy of LFTJ and thus preserving consistent enumeration procedure. Unlike LFTJ, APEX derives a lookup key from the bindings of a variable's dependency variables and directly retrieves the corresponding candidate list from the $\mathcal{G}$, without performing any intersection. The traversal is parallelized by partitioning the domain of the first variable into n disjoint slices, each processed by a separate thread. Since this phase mainly involves enumeration and backtracking rather than intersection computation, potential workload imbalance has limited impact on performance.

The function `CPGTraverse` first initializes the solution list **R** (Line 15) and launches n parallel traversals (Lines 16–18). Each thread starts with an empty binding set σ and invokes `Traverse` from the root variable. The candidate list of the first variable is partitioned so that each thread processes a disjoint slice (Lines 8–9). After all parallel traversals finish, the aggregated solution list **R** is returned (Line 19).

The recursive function `Traverse` explores CPG step by step. At each call, it first checks whether all variables have been assigned. If the index idx reaches the end of Φ , the current binding set σ is inserted into **R** (Lines 2–3). Otherwise, the next variable v is selected from Φ (Line 5). The binding combinations of v 's in-neighbors in $\mathcal{G}$ are constructed (Line 6) and used to retrieve the candidate list $cands$ of v from $\mathcal{G}$ (Line 7). For each value in $cands$, σ is extended with a binding for v (Line 11), followed by recursion on the next variable (Line 12). After recursion returns, the binding of v is removed from σ to restore the partial state (Line 13). This procedure ensures that all solutions are enumerated.

Storage Overhead. Let k be the number of variables in the query. For each variable v_i , let $d_i = |\text{In}_{\mathcal{G}}(v_i)|$, n_i be the number of candidate lists stored for v_i , and ℓ_i be the average candidate list size at v_i . The overall space complexity is $O(\sum_{i=1}^k n_i \ell_i + \sum_{i=1}^k n_i d_i)$.

The number of generated bindings in LFTJ is asymptotically bounded by the query's worst-case output size. Therefore, the first term $\sum_{i=1}^k n_i \ell_i$, which represents the total number of variable bindings, is also bounded. For the second term, since d_i is small in practice, $\sum_{i=1}^k n_i d_i$ remains modest. Therefore, the storage overhead of CPG is acceptable in practice.

V. VARIABLE SELECTION

A. Variable Ordering as Markov Decision Process

The generation of a variable ordering can be naturally formulated as a Markov Decision Process (MDP) [17], defined by the tuple $\langle S_0, \mathcal{S}, \mathcal{A}, \mathbb{P}, R \rangle$, where $\mathcal{S}$ is the state space, S_0 is the initial state, $\mathcal{A}$ is the action space, $\mathbb{P} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is the transition function, and $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function. At step t , the state $S_t \in \mathcal{S}$ includes the current processed variables Φ_t , the query graph Q , and the current CPG $\mathcal{G}_t$. The action space $\mathcal{A}_t$ is the unprocessed variables. An action selects a variable to extend Φ_t and updates $\mathcal{G}_t$, then processes this variable. Finally, the reward R_t is derived from runtime feedback. On this basis, we can leverage reinforcement learning (RL) techniques to learn a variable order [33].

State. At step t , we represent the state S_t as a derived graph encoding, which is prepared for the GCN encoder. Specifically, S_t includes two components: (1) vertex feature matrix M_t and (2) edge feature matrix E . Fig. 5 illustrates them with an example. More specifically:

(1) **Vertex Feature Matrix.** The vertex feature matrix $M_t = [m_t^{(1)}, \dots, m_t^{(i)}, \dots, m_t^{(n)}]$ captures the current status of each variable at step t , where $n = |V(Q)|$ is the number of variables (i.e., vertices) in query graph Q . Each column $m_t^{(i)} \in \mathbb{R}^{3 \times 1}$ represents the feature vector of variable v_i , capturing three key attributes that characterize the current evaluation context:

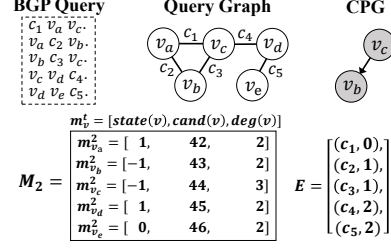


Fig. 5. Vertex and Edge Feature Example.

- Vertex status** ($\text{status}(v_i)$) indicates whether v_i is selectable. If $v_i \in \Phi_t$, its status is -1 . If v_i is a neighbor of a variable in Φ_t in Q , its status is 1 ; otherwise, it is 0 .
- Candidates count** ($\text{cand}(v_i)$) is derived from the current CPG $\mathcal{G}_t$. If $v_i \in \Phi_t$, we use the exact candidates count $\text{size}(v_i)$; otherwise, we use the estimated value $\text{est}(v_i)$.
- Degree** ($\text{deg}(v_i)$) is simply the degree of v_i in Q .

(2) **Edge Feature Matrix.** Each edge in Q corresponds to a triple pattern $t = (s, p, o)$. For each edge e , we extract a tuple $(\text{label}(e), p_e)$, where $\text{label}(e)$ denotes the constant ID, and $p_e \in \{0, 1, 2\}$ indicates whether the constant appears as the subject, predicate, or object. All such tuples collectively form the edge feature matrix E .

Action. At step t , the action space $\mathcal{A}_t$ consists of unprocessed neighbors of variables in Φ_t . For initialization, $\mathcal{A}_0 = \{v \in V(Q) \mid \text{deg}(v) > 1\}$. If at step t the action space is empty, i.e., $\mathcal{A}_t = \emptyset$, we fall back to $\mathcal{A}_t = \{v \mid \text{status}(v) = 0\}$. The selected action v' extends the ordering as $\Phi_{t+1} = \Phi_t \cup \{v'\}$.

Reward. The reward for a selected action consists of three components. Two of them are defined relative to the baseline variable order Φ^{base} . This baseline is a dynamic greedy ordering generated by a simple heuristic: (i) expand only from neighbors of previously selected variables; (ii) among these, select the variable with the highest degree; (iii) if multiple variables have the same degree, the one with the smallest $\text{cand}(v_i)$ is chosen. For example, in Fig. 5, we first select v_c (highest degree), then choose among its neighbors v_a , v_b , and v_d , not v_e . Since these variables have the same degree, we select v_a , which has the smallest candidates count. Expanding from neighbors prevents Cartesian products [17]. In addition, prioritizing higher-degree variables, which are more constrained, reduces candidate list sizes and search space. This leads to more efficient query execution.

- Immediate reward** (R_t^{size}). R_t^{size} measures the reduction in candidates count relative to $\Phi^{\text{base}}[t]$, encouraging the policy to minimize intermediate results.
- Time reward** (R_t^{time}). R_t^{time} measures the reduction in processing time for the variable selected at step t relative to $\Phi^{\text{base}}[t]$. If $R_t^{\text{size}} = 0$, a small negative value is assigned to R_t^{time} to encourage faster processing.
- Validity reward** (R_t^{val}). R_t^{val} rewards valid actions. A positive reward is given if the variable with the highest predicted probability lies within $\mathcal{A}_t$ before masking; otherwise, a larger negative penalty is applied.

Both R_t^{size} and R_t^{time} are normalized. Therefore, the overall

step-wise reward is formulated as:

$$R_t = \lambda_1 \cdot \hat{R}_t^{\text{size}} + \lambda_2 \cdot \hat{R}_t^{\text{time}} + \lambda_3 \cdot R_t^{\text{val}}, \quad (2)$$

where λ_1 , λ_2 and λ_3 are coefficients that control the impact of each reward component during training.

B. Policy Network

We use a GCN encoder to capture the structure of the query graph. Unlike per-node MLPs, the GCN aggregates multi-hop neighbor information, reflecting how variables and edge labels jointly influence selectivity. This is crucial because variable ordering decisions depend not only on local statistics of individual variables, such as degree or candidate value count, but also on the relationships between variables and the query graph structure. Even variables that are not directly connected can impose indirect constraints through intermediate variables. As a result, effective ordering often relies on multi-hop structural relationships rather than isolated local features.

GCNs naturally perform message passing over the query graph, allowing each variable representation to aggregate information from its multi-hop neighbors and explicitly model structural dependencies. This enables the policy to reason about how constraints jointly affect selectivity and execution cost across the query graph. In contrast, an MLP based solely on local statistics lacks access to relational structure and thus cannot effectively capture cross-variable dependencies. The GCN is permutation-invariant, can scale to queries of varying sizes, and adds small inference cost. Compared to attention-based encoders, it provides a simpler and more efficient trade-off between representation quality and computation.

Building on this motivation, edge information from $\mathcal{Q}$ is fed into the GCN. The node feature matrix M_t and E are jointly processed by the policy network. Each edge feature $(\text{label}(e), p_e) \in E$ is embedded and passed through a MLP to produce a scalar edge weight $w_e \in (0, 1)$. The edge-weight MLP is shallow and processes edges independently, leading to linear complexity in the number of edges. Node features in M_t are then combined with these weights and propagated through two weighted GCN layers to obtain contextualized vertex representations X_q . Since the GCN consists of only two layers, its computation scales linearly with the number of edges and the hidden dimension. Prior production systems, such as *Stage* [34] in Amazon Redshift, have deployed deeper GCN models in latency-sensitive scenarios without inference becoming a bottleneck. Finally, a lightweight MLP projects X_q into selection logits, representing action probabilities over variables. As a result, the policy network remains lightweight while still effectively capturing key features in the query graph.

Invalid Action Masking. To improve convergence, we apply action masking at each step [35]. Let $m_t(v) = 1$ if $v \in \mathcal{A}_t$ and 0 otherwise. Given logits ℓ , we form masked logits as

$$\tilde{\ell}_v = \begin{cases} \ell_v & \text{if } m_t(v) = 1 \\ 0 & \text{otherwise} \end{cases}$$

and obtain the action probability distribution $\pi_\theta(v \mid s_t)$. During training, vertices are sampled from this distribution to

encourage exploration; during inference, we select the variable by $v' \leftarrow \arg \max_v \pi_\theta(v \mid s_t)$.

Policy Training. The policy is trained with Proximal Policy Optimization (PPO) in an actor-critic setup [36]. Advantage estimates $\hat{A}_t$ are computed via Generalized Advantage Estimation (GAE) [37], and $\hat{R}_t$ denotes the discounted return from state s_t . An entropy bonus encourages exploration, and gradient clipping is applied for training stability. The PPO surrogate objective is defined as

$$J(\theta, \phi) = \mathbb{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right] - c_v \mathbb{E}_t \left[(V_\phi(s_t) - \hat{R}_t)^2 \right] + c_e \mathbb{E}_t \left[H(\pi_\theta(\cdot \mid s_t)) \right],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$ is the probability ratio of the new and old policies, ϵ is the clipping parameter limiting $r_t(\theta)$, $V_\phi(s_t)$ is the value function parameterized by ϕ , c_v and c_e weight the value loss and entropy bonus, respectively, and $H(\pi_\theta(\cdot \mid s_t))$ is the policy entropy at state s_t .

Risk-Aware Decision Strategy. We further introduce a conservative decision-making strategy that explicitly accounts for the learned policy's uncertainty inspired by [38], [39].

Our variable selector follows an *actor-critic* architecture, in which the policy is jointly determined by an *actor network* and a *critic network*. In particular, the critic's output directly influences the final action selection. Then, for each candidate variable v_i , the critic network outputs a scalar score (higher is better). To capture predictive uncertainty, we further apply Monte Carlo dropout and perform multiple stochastic forward passes through the critic network, obtaining an empirical mean μ_{v_i} and standard deviation σ_{v_i} . At runtime, the next variable is selected by maximizing $\mu_{v_i} - f_s \cdot \sigma_{v_i}$, where f_s is a parameter that denotes the degree of risk awareness. Note, although this approach requires multiple inferences, dropout is enabled only in the critic head, which is a lightweight MLP, while the shared representation is computed once and reused.

VI. PARALLEL JOIN EXECUTION

A. Generating Variable Bindings

After selecting the next variable v' , we enumerate binding combinations of the already processed variables by traversing the CPG $\mathcal{G}$, and use these combinations to perform intersections for v' . A naive approach traverses $\mathcal{G}$ strictly following the global variable order Φ . However, when the variables in $\Phi \setminus \{v'\}$ span multiple connected components of $\mathcal{G}$, such a traversal implicitly induces Cartesian products across independent components.

Algorithm 3 avoids this by exploiting the fact that variables within the same connected component are interdependent, whereas variables from different components are independent. It first initializes a list of binding groups $\mathbf{B}$ (Line 2) and computes the connected components of $\mathcal{G}$ excluding v' (Line 3).

For each component C (Lines 4–7), the algorithm derives a component-wise order Φ_c (Line 5). Specifically, it sorts the variables in C by their positions $\text{pos}(\cdot)$ in Φ , which preserves the relative order induced by Φ . The traversal is thus restricted

Algorithm 3: Generating Variable Bindings.

Input: Query graph Q , CPG $\mathcal{G}$, variable order Φ , next variable v' , number of threads n

Output: Variable binding groups B

1 Function

GenerateVariableBinding($Q, \mathcal{G}, \Phi, v', n$):

```
2  B  $\leftarrow []$ 
3  comps  $\leftarrow$  connected components in  $\mathcal{G}$  excluding  $v'$ 
4  for  $C \in \text{comps}$  do
    // Sort variables in  $C$ 
5     $\Phi_c \leftarrow [u \mid u \in V(C), \text{sorted by } \text{pos}(u)]$ 
    // Enumerate in this component
6     $B_c \leftarrow \text{CPGTraverse}(\mathcal{G}, \Phi_c, n)$ 
7     $B \leftarrow B \parallel [B_c]$ 
8  return B
```

to C . It then invokes $\text{CPGTraverse}(\mathcal{G}, \Phi_c, n)$ to enumerate all valid binding combinations within C , producing B_c (Line 6), and appends B_c to B as an independent binding group (Line 7).

Finally, the algorithm returns B (Line 8), which is later combined via Cartesian products and used for intersections in the parallel join phase (Section VI-B).

B. Parallel Join

Building on the grouped binding combinations B from Algorithm 3, where $B = [B_1, \dots, B_j]$, each element in B represents a list of binding combinations. The final binding combinations are computed via Cartesian products, and the intersections for variable v' are performed in parallel.

To balance the workload across threads, the largest binding group B_i is partitioned into n subgroups, $B_i^1, \dots, B_i^n$, where n is typically several times more than the number of execution threads. These subgroups are then enqueued into a shared task queue. Then, execution threads dynamically fetch group processing tasks from this queue, and idle threads proactively acquire additional work. Under this execution model, the computation cost might differ significantly across threads due to different intermediate result sizes in each group. However, threads with lighter workloads naturally take on more tasks, leading to balanced execution across threads.

Within the each thread, its assigned m -th portion B_i^m replaces B_i in the original binding groups, forming thread-specific binding groups $[B_1, \dots, B_i^m, \dots, B_j]$. The Cartesian product among these groups is then enumerated in the thread to generate its portion of the final binding combinations. Each element in resulting combination $[b_1, \dots, b_j]$ corresponds to the bindings of some neighbors of v' in Q . These partial combinations are merged into a complete binding combination σ , representing bindings of the dependency variables of v' . Using σ and corresponding edge labels, the algorithm retrieves the intersection lists from the tries for each neighbor in Q .

These lists are subsequently passed to LEAPFROG JOIN to compute the intersection and produce the candidate list. If the candidate list is non-empty, it is stored at vertex v' in

$\mathcal{G}$ together with the binding combination of its in-neighbors. The stored results are later consumed by the traversal routine (Algorithm 2), completing the loop between binding generation and intersection execution. Enumeration continues until all combinations assigned to the thread have been processed.

C. Redundancy Optimization

In the previous section, we introduce how to parallelize LFTJ by enumerating variable binding combinations. However, this approach inevitably introduces two forms of redundancy, i.e., traversal redundancy and lookup redundancy. To this end, we propose a dependency-source detection method to address traversal redundancy and a filter to address lookup redundancy.

(1) Traversal Redundancy. We observe that a full traversal of the CPG $\mathcal{G}$ is often unnecessary. For instance, if the current variable v' has only one in-neighbor s in $\mathcal{G}$, i.e., $\text{In}_{\mathcal{G}}(v') = \{s\}$. In this case, regardless of the input bindings to vertex s , the output of s is the union of all its candidate lists. Therefore, the evaluation of v' can directly use the output of s as its input, without a full traversal of $\mathcal{G}$ in the global order Φ .

More generally, the candidate lists for a variable v' depend only on its in-neighbors $\text{In}_{\mathcal{G}}(v')$. Thus, we can identify the source of dependency for v' via the backward paths induced by the in-neighbors. By recursively following these dependencies, we identify the *dependency source* for v' as the vertex that lies on every backward path (from v' following in-edges) in $\mathcal{G}$:

$$\mathcal{P}_{\uparrow}(v') := \left\{ (v_0, \dots, v_{\ell}) \mid \begin{array}{l} v_0 = v', \ell \geq 0 \\ v_{i+1} \in \text{In}_{\mathcal{G}}(v_i), 0 \leq i < \ell \\ \text{In}_{\mathcal{G}}(v_{\ell}) = \emptyset \end{array} \right\}. \quad (3)$$

That is, $\mathcal{P}_{\uparrow}(v')$ represents all maximal backward paths from v' to sources (vertices with no in-neighbors). We then capture the set $\mathcal{M}$ of vertices that appear on *every* backward path from v' . Since the subgraph of $\mathcal{G}$ that we traverse is assumed to be connected (as guaranteed by Algorithm 3), $\mathcal{M}$ is always non-empty. We select the vertex in $\mathcal{M}$ that is latest in the global order Φ , i.e., $s := \arg \max_{u \in \mathcal{M}} \text{pos}(u)$. Then the evaluation of v' only needs to traverse the suffix $[\text{pos}(s), \Phi_{\max}]$, pruning the traversal to eliminate redundant work.

Moreover, vertex s may contain multiple candidate lists, many overlapping. To reduce redundancy, we compress s into a temporary vertex s' , which contains only a single input-output pair. The input is an empty binding set, and its candidate list is all distinct values in candidate lists stored in s .

We then temporarily replace vertex s with s' in $\mathcal{G}$ and restrict traversal to the suffix $[\text{pos}(s), \Phi_{\max}]$. This ensures that only the relevant portion of $\mathcal{G}$ is explored. The optimized traversal procedure enhances CPGTraverse in Algorithm 3.

(2) Lookup Redundancy. Lookup redundancy arises from two main sources. **(i)** When retrieving candidate lists from a $\mathcal{G}$ vertex, the result may be empty. During intersection, binding combinations are used to fetch intersection lists, but empty intersection results are not stored in the vertex. As a result, CPG traversal may still generate and query binding combinations that lead to empty results, introducing unnecessary overhead. **(ii)** When dependency source detection is employed, candidate values of a variable that have already been

eliminated may be reselected and processed again, leading to redundant intersection operations. This issue is particularly pronounced for high-degree variables.

To address this issue, we attach a Bloom filter (BF) to certain vertices in $\mathcal{G}$, which serves as a validity indicator. If a value appears in BF, it is considered valid. Then, the BF allows us to track the validity of values in candidate lists for the dependency source s of the current variable v' . Specifically, we obtain all binding combinations of variables in the order suffix $[\text{pos}(s), \Phi_{\max}]$, which includes the dependency source s . If s has degree greater than one in Q and an unprocessed neighbor other than v' , a BF is constructed for vertex s during the intersection phase (Section VI-B). Specifically, we obtain all binding combinations of variables in the order suffix $[\text{pos}(s), \Phi_{\max}]$, which includes the dependency source s . If an intersection result is non-empty, the binding value of s is inserted into the filter, marking it as valid. Later, during traversal of $\mathcal{G}$ (Algorithm 2), any value in the candidate lists of s not contained in the BF is skipped directly. This reduces redundant lookups of invalid candidates and eliminates unnecessary work during traversal and intersection.

D. Time Complexity Analysis

The time complexity of LFTJ is $O((\sum_{i=0}^{k-1} C_i) \log N_{\max})$, where $N_{\max}$ is the maximum relation size and C_i is the *sum-min count*, an upper bound on the number of bindings generated for v_i . We analyze the complexity of APEX relative to LFTJ under a variable order $\Phi = [v_0, \dots, v_{k-1}]$, assuming dependency-source detection and Bloom filters are disabled.

Intersection Equivalence. We show by induction that APEX and LFTJ perform identical intersection operations.

Base case ($i = 0$). Both algorithms start from the empty prefix binding and perform a single intersection for v_0 over identical input lists, resulting identical results.

Inductive step. Assume that for variables $v_0, \dots, v_{i-1}$, both algorithms enumerate the same prefix bindings and produce identical intersection results. This holds because both algorithms use DFS, follow the same variable order, and operate on the same intersection results. For each such prefix binding, both APEX and LFTJ perform exactly one intersection for v_i over identical lists, producing identical results.

By induction, APEX and LFTJ execute the same intersection operations for all variables. Therefore, the total intersection cost of APEX is $T_{\text{join}} = O((\sum_{i=0}^{k-1} C_i) \log N_{\max})$.

Overhead and Total Complexity. APEX introduces additional overhead due to CPG traversal and inserting results into it, both proportional to the total number of candidate values. Traversing the CPG over k variables incurs a cost of $T_{\text{traverse}} = O(k \sum_{i=0}^{k-1} C_i)$, while storing intersection results incurs $T_{\text{store}} = O(1 + \sum_{i=1}^{k-1} C_{i-1})$.

Since the number of variables k is treated as a constant, the overall time complexity of APEX is given by the sum of T_{join} , T_{traverse} , and T_{store} , which is $O((\sum_{i=0}^{k-1} C_i) \log N_{\max})$. This shows that APEX preserves the worst-case asymptotic complexity of standard LFTJ. Due to space limitation, the complete proof can be found at [40].

TABLE I. Sizes of Dataset and Tries (M = Million).

Dataset	Edges (M)	Nodes (M)	Predicates	Trie Size (GB)
Wikidata	1,257.17	364.79	8,604	11.65
WatDiv500M	549.46	49.78	86	3.64
Wikidata-sub	82.92	54.25	2,101	0.87
DBpedia	1,040.35	216.32	68,687	8.94
YAGO 2.3.0	109.89	54.35	96	0.96

VII. EXPERIMENTAL EVALUATION

We implement APEX using C++ (GCC 12.3.0) for data loading and query execution, and Python (3.8.10) for generating variable orders. Our implementation and all the queries used in our experiments can be found in [40]. To support efficient WCOJ, APEX maintains two permutations of trie structures (subject–predicate–object and object–predicate–subject) for the graph data. To satisfy the *persistence* requirements of database systems, we store tries on disk.

A. Experimental Setup

Datasets and Queries. We evaluate query engines on a variety of knowledge graph datasets, covering both synthetic and real-world datasets and queries. Table I summarizes the data graphs used in this section and their trie index sizes, indicating that APEX’s trie index is manageable in practice.

Based on these knowledge graphs, we use the following query workloads. **Wikidata Graph Query Benchmark (WD-Bench)** [41] is derived from the Wikidata knowledge graph (Truthy dump, 2021-06-23) and provides real-world queries; from the original 681 BGP queries, we retain 158 queries with more than three triple patterns. **The Waterloo SPARQL Diversity Test Suite (WatDiv)** [42] is a synthetic dataset. We generate Linear (L), Star (S), Snowflake-shaped (F), and Complex (C) queries using its data and query generators, and additionally include queries from the stress mode. After retaining only queries with more than three triple patterns, we obtain a total of 169 queries, referred to as **WatDiv-S**. **The Wikidata Graph Pattern Benchmark (WGPB)** [13] is based on a Wikidata subgraph (Truthy dump, 2018-11-15) and consists of 17 query patterns covering both cyclic and acyclic structures. Each pattern includes 50 queries, categorized into six types: n -way Join (J), Path (P), Square (S), Star (T), Star Inverse (TI), and Triangle (Tr). **The Large-scale Complex Question Answering Dataset (LC-QuAD)** [43] targets DBpedia (04-2016) and provides 5,000 queries, from which we select 1,000 queries with higher structural complexity. **YAGO** [44] is a real-world knowledge graph; we use YAGO 2.3.0 and evaluate it with the query sets (y1–y4) defined in prior work [45], [46].

Baselines. We compare APEX with representative systems, including ADOPT [17], Umbra [11], Diamond [14], Kùzu [9], Neo4j [1], MillenniumDB [10], POLAR [47], and GLO [48].

The following systems support WCOJ algorithms. Umbra (2022-11-03) is a database engine with native parallel multiway joins via morsel-driven execution. Diamond is a variant of Umbra optimized for diamond-shaped queries. Kùzu (0.11.2) is a recent graph database with vectorized execution and morsel-driven parallelism; we use its in-memory mode.

ADOPT is an in-memory WCOJ system that processes non-overlapping variable cubes in parallel and adapts variable orders via reinforcement learning. MillenniumDB is a graph database that supports both binary joins and LFTJ, selecting between them during query planning.

The following systems do not support WCOJ algorithms but are included for a comprehensive comparison. Neo4j (v5.20.0) is a widely used graph database employing native graph storage, index-free adjacency, and optimizations for efficient graph query processing. POLAR is a non-invasive adaptive join processing built evaluated on DuckDB 1.0.0 that augments a left-deep join pipeline with multiple alternative join orders and dynamically routes tuples based on runtime feedback. GLO is a learning-based join order optimizer built on PostgreSQL 13.0, designed for generalization using statistic-enhanced query representations and value-based deep reinforcement learning.

Setup. During evaluation, all systems are configured to run in memory to avoid the impact of I/O. All systems are configured to use 32 threads, and indexes are prepared for each system. To assess the quality of variable orders and execution efficiency, the reported execution time excludes inference time for all systems, including APEX and other baselines that incur additional planning or inference overhead. Each experiment is run three times, and we report the average execution time. Experiments are conducted on a server running Ubuntu 20.04, equipped with an AMD EPYC 75F3 CPU (32 physical cores, 2.95 GHz), 128 GB RAM, a 1024 GB NVMe SSD, and an NVIDIA GeForce RTX 3090 GPU.

B. Runtime Performance

We first evaluate all systems across all datasets. For each query set, we uniformly sample 50% of the queries for training and the remaining 50% for testing. The results are reported in Fig. 6, Fig. 7, and Fig. 8. Since APEX requires a sufficient number of queries to train the policy, datasets such as WatDiv and YAGO contain too few queries to support effective learning. Therefore, the performance of APEX on these datasets is not reported in Fig. 8. Instead, we include a variant, APEX-DG, which replaces the learned selector with the dynamic greedy ordering described in Section V-A, enabling a comprehensive comparison. ADOPT is excluded from WDBench and LC-QuAD because it is an in-memory system and the data graph used in these datasets exceed the available memory of our experimental platform.

The results in Fig. 6, Fig. 7, and Fig. 8 show that APEX and APEX-DG perform consistently well across most datasets. The only exception is the y_3 query on YAGO, where APEX-DG performs worse. On WDBench, APEX achieves the best overall performance, demonstrating the effectiveness of the learned selector on diverse and complex workloads. In contrast, the performance gain of APEX over APEX-DG on WatDiv is limited, as many queries contain highly skewed variables for which simple degree-based ordering is already near-optimal. Similarly, on LC-QuAD, most queries involve only a few variables, reducing the benefit of learning-based ordering. On WGPB, APEX outperforms APEX-DG mainly on J, P, and

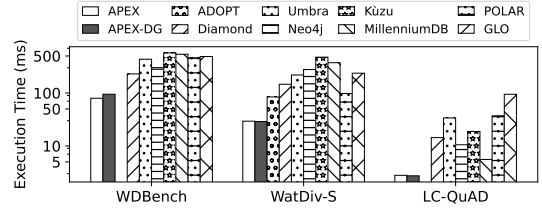


Fig. 6. Runtime on WDBench, WatDiv-S, and LC-QuAD.

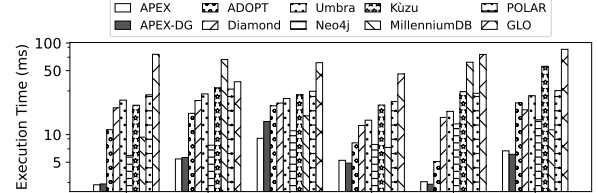


Fig. 7. Runtime on WGPB.

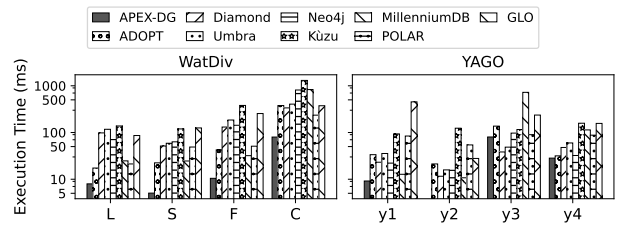


Fig. 8. Runtime on WatDiv and YAGO.

S query types. For T and TI queries, a single high-degree variable dominates, making greedy ordering effective, while for Tr queries, similar variable degrees and small query sizes limit the benefit of learning-based reordering.

Fig. 9 reports the average inference time per query by component. The GCN encoder dominates inference time, whereas the edge-weight MLP and actor head incur minimal overhead. The higher inference cost on WatDiv-S is due to more frequent policy invocations caused by higher-degree variables.

Fig. 10 reports the performance variance of APEX across five runs with different random seeds. We observe that the performance variance is relatively small, indicating *stable* performance. This stability arises from the risk-aware safeguard, which accounts for predictive uncertainty in the critic's estimates and suppresses high-risk variable selections, thereby reducing sensitivity to randomness and runtime feedback.

C. Candidate Propagation Graph Overhead

We evaluate the overhead of the Candidate Propagation Graph (CPG) in terms of traversal cost and memory usage using all queries across all datasets. Fig. 11 reports the execution time breakdown across different query stages. Overall, CPG traversal accounts for only a small fraction of the total execution time, averaging 6.19% in internal traversal (IT) and 13.33% in the final traversal (FT). The FT stage closely resembles the traversal and backtracking procedure in LFTJ. Consequently, the additional cost introduced by APEX mainly arises from IT. Across all datasets, IT is consistently faster

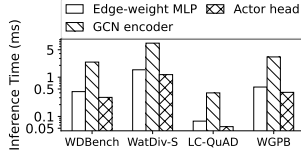


Fig. 9. Average Policy Inference Time Per Query.

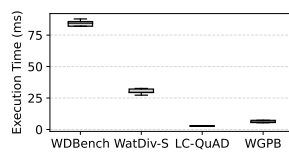


Fig. 10. Performance Variance Across Seeds.

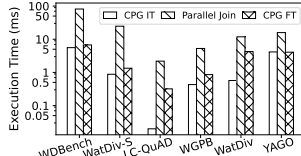


Fig. 11. Execution Breakdown.

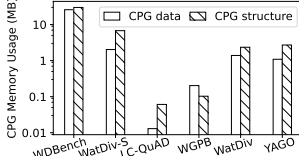
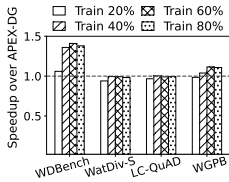
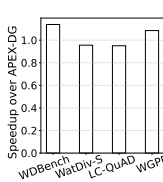


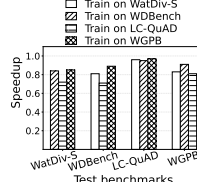
Fig. 12. CPG Memory Usage.



(a) Varying Training Ratios.



(b) Distribution Shift.



(c) Dataset Transfer.

Fig. 13. Generalization Analysis of the Learned Policy.

than FT, indicating that APEX spends most of its execution time on actual join computation rather than on CPG traversal.

Fig. 12 summarizes the CPG memory usage across datasets. The average peak CPG memory footprint remains at the megabyte scale, with a maximum of 29.84 MB, demonstrating that its memory consumption is manageable in practice.

D. Generalization Performance

We evaluate generalization along three different conditions. First, to study the effect of sparse or potentially biased training data. Specifically, we use training ratios of 20%, 40%, 60%, and 80%, reserving the remaining queries for testing, and report the speedup of APEX relative to APEX-DG. Fig. 13a shows that APEX underperforms when trained with only 20% of the queries. Once the training ratio reaches 40%, APEX achieves better performance than APEX-DG. This suggests that while limited training data harms performance, the RL-based variable ordering generalizes well with moderate data. However, using an 80% training ratio on WDBench and WGPB results in a performance drop compared to 60%, likely due to overfitting and a smaller, less stable test set.

Second, to assess robustness under query distribution shifts, we partition queries within each dataset by query patterns, training the policy on one subset and evaluating it on disjoint, unseen patterns. Specifically, for WDBench, we train on 91 star-shaped queries and test on 67 tree- and cyclic-shaped queries; for WatDiv-S, we train on 73 tree-shaped queries and test on 96 star-, path-, and cyclic-shaped queries; for WGPB, we train on star-shaped patterns (J, T, and TI; 400 queries) and test on path and cyclic patterns (P, S, and Tr; 450 queries); and for DBpedia, we train on 648 star-shaped queries

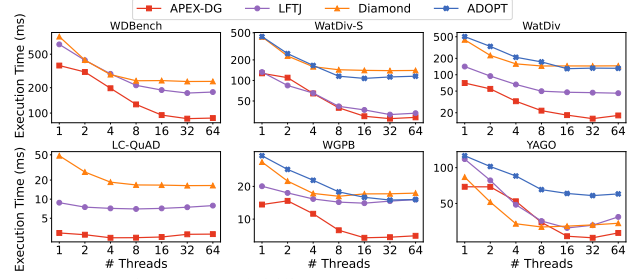


Fig. 14. Scalability w.r.t. Thread Count.

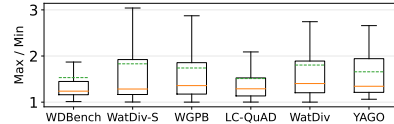


Fig. 15. Evaluation of Load Balancing.

and test on 352 path-shaped queries. Fig. 13b reports the results. While training on limited types of patterns degrades performance compared to mixed-pattern training, APEX outperforms APEX-DG on WDBench and WGPB, indicating that the learned policy generalizes to unseen query types.

Third, we evaluate generalization to unseen datasets via cross-dataset experiments by training the policy on one of WatDiv-S, WGPB, WDBench, or LC-QuAD and testing it on the others. Although performance degrades compared to in-dataset training, APEX remains stable across all transfer settings except training on LC-QuAD. This degradation likely arises from the simplicity of LC-QuAD queries, which provides insufficient supervision for the policy network and results in poorer performance on more complex target datasets.

E. Scalability and Load Balancing

We next evaluate the scalability of APEX by varying the number of threads from 1 to 64 on all datasets and queries (Fig. 14). We compare APEX-DG with Diamond, ADOPT, and LFTJ. For LFTJ, we parallelize execution by partitioning the domain of the first variable and use the same variable ordering as APEX-DG to ensure a fair comparison. Other systems are excluded from this experiment, as their performance with 32 threads was unsatisfactory.

As shown in Fig. 14, APEX-DG achieves good speedups on most datasets except LC-QuAD. This is because queries in LC-QuAD generally involve relatively small workloads, limiting the benefits of multi-threading. On other datasets, APEX-DG may exhibit limited speedup when using only two threads. In particular, on WGPB and YAGO, it can even show slightly worse performance when using a small number of threads, as these datasets involve smaller datasets where parallelization overhead dominates. However, as the number of threads increases, APEX-DG consistently achieves the lowest runtime across most datasets.

We further measure the per-thread execution time of APEX-DG when processing each variable under 32 threads. Fig. 15 reports, for each variable, the ratio between the maximum and minimum execution time across threads. The results show that

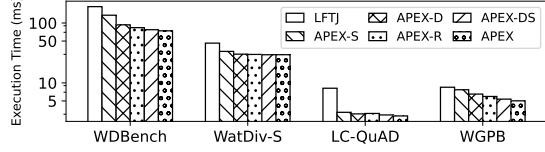


Fig. 16. Results of Ablation Study.

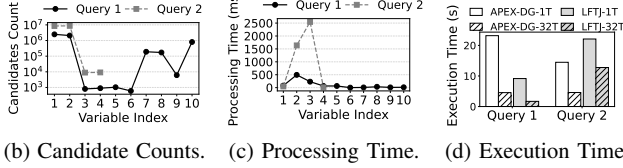
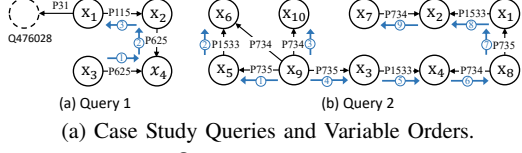


Fig. 17. Results of Case Study.

this ratio remains relatively small across variables, indicating limited thread skew and effective load balancing in APEX-DG.

F. Ablation Study

We progressively enable the components of APEX to quantify their individual contributions: (1) **LFTJ** executes the original LFTJ with a static greedy variable ordering, serving as the baseline. The static greedy ordering means the variable order is determined before the query starts, based on the degrees of the variables and estimated number of candidate values, and remains fixed throughout execution. Parallel execution is achieved by partitioning the domain of the first variable across threads. (2) **APEX-S** introduces the BFS/CPG execution model with static ordering; (3) **APEX-D** extends APEX-S with dynamic greedy ordering which is described in Section V-A; (4) **APEX-R** further incorporates an RL-based variable selector; (5) **APEX-DS** enables dependency-source detection; and (6) **APEX** is the full system, which additionally applies Bloom-filter-based pruning.

Fig. 16 illustrates how these components affect performance. Introducing the BFS/CPG execution model (APEX-S) substantially improves performance over LFTJ on datasets with large data graphs, such as WDBench, WatDiv-S, and LC-QuAD. On WGPB, however, the benefit is limited and APEX-S can even be slower than LFTJ, as the queries involve smaller graphs or simpler structures with limited parallelism, making CPG overhead harder to amortize.

Building on APEX-S, dynamic greedy ordering (APEX-D) consistently yields further improvements. In contrast, the RL-based selector (APEX-R) performs worse than APEX-D on WatDiv-S and LC-QuAD, where simple heuristics already handle high-degree variables effectively and the small number of variables limits the benefit of learning-based ordering.

Finally, enabling dependency-source detection and Bloom-filter-based pruning (APEX-DS and full APEX) provides additional gains on more complex queries, particularly on WDBench and WGPB.

G. Case Study

APEX is less effective for queries with limited parallelism or very large intermediate results at early stages. In these cases, many candidates are materialized early and heavily pruned later, which increases memory usage and reduces the benefit of parallel execution. In contrast, when join result sizes shrink after a few variables, APEX can effectively exploit parallelism and amortize its overhead.

Fig. 17a shows two representative queries used in our case study, with blue arrows indicating the variable processing order. Fig. 17d reports the execution time of APEX-DG and LFTJ under 1 and 32 threads, while Figures 17b and 17c show the candidate counts and per-variable processing time for APEX-DG under 32 threads.

For Query 1, the selected variable ordering processes high-cardinality variables early (Fig. 17b). As a result, CPG materialization incurs a large memory footprint of 821.64 MB, and CPG traversal accounts for 36.09% of the total execution time. Because the first two variables have many candidate values, many intersections are required at the second and third variables, concentrating most computation on these stages (Fig. 17c). Given that the query contains only four variables, this query causes only mild thread skew in LFTJ. As a result, LFTJ achieves better performance and scalability than APEX-DG, which additionally incurs CPG overhead (Fig. 17d).

In contrast, Query 2 involves more variables, many of which have large candidate value counts (Fig. 17b). As a result, CPG materialization consumes 221.14 MB of memory, and CPG traversal accounts for 62.41% of the total execution time. Although large candidate value counts at later variables indicate frequent intersections, the processing time of these variables in APEX-DG remains low, showing that APEX-DG effectively exploits parallelism (Fig. 17c). For LFTJ, partitioning work solely based on the first variable fails to evenly distribute the computation of subsequent variables across threads, resulting in severe thread skew. As a result, despite the overhead of CPG traversal, APEX-DG achieves significantly lower overall execution time than LFTJ. As shown in Fig. 17d, LFTJ exhibits only limited speedup as the number of threads increases.

VIII. CONCLUSION

This paper presents APEX, an adaptive variable-wise parallel execution for worst-case optimal joins on graph queries. It organizes data through a candidate propagation graph to enable parallel BFS-style execution of LFTJ. APEX further incorporates a learning-based variable selector and redundancy-reduction techniques. Experimental results demonstrate that APEX achieves efficient and scalable query processing compared with the existing systems.

Acknowledgement. This work was supported by the NSFC (No. 62362015), Guangxi Key R&D Program (No. Gui Ke AB24010160), Innovation Project of GUET Graduate Education (No. 2025YCX056), National Key R&D Program of China (No. 2024YFC3308102), and NSFC (No. 62461146205, 92270202, 62202171).

IX. AI-GENERATED CONTENT ACKNOWLEDGEMENT

Portions of this manuscript were refined using ChatGPT to improve linguistic clarity and grammatical accuracy. All AI-assisted content was thoroughly reviewed and verified by the authors to ensure its technical correctness and originality.

REFERENCES

- [1] “Neo4j graph database,” <https://neo4j.com>.
- [2] A. Deutsch, Y. Xu, M. Wu, and V. Lee, “Tigergraph: A native mpp graph database,” *arXiv preprint arXiv:1901.08248*, 2019.
- [3] H. Q. Ngo, C. Ré, and A. Rudra, “Skew strikes back: new developments in the theory of join algorithms,” *SIGMOD Rec.*, vol. 42, no. 4, p. 5–16, Feb. 2014.
- [4] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, “Worst-case optimal join algorithms,” *J. ACM*, vol. 65, no. 3, pp. 16:1–16:40, 2018.
- [5] T. L. Veldhuizen, “Triejoin: A simple, worst-case optimal join algorithm,” in *Proc. 17th International Conference on Database Theory (ICDT)*, Athens, Greece, March 24–28, 2014. OpenProceedings.org, 2014, pp. 96–106.
- [6] O. Kalinsky, A. Hogan, O. Mishali, Y. Etsion, and B. Kimelfeld, “Exploration of knowledge graphs via online aggregation,” in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9–12, 2022*. IEEE, 2022, pp. 2695–2708.
- [7] S. Park, S. Oh, and M.-S. Kim, “cumatch: A gpu-based memory-efficient worst-case optimal join processing method for subgraph queries with complex patterns,” *Proc. ACM Manag. Data*, vol. 3, no. 3, Jun. 2025.
- [8] J. Fuentes-Sepúlveda, A. Gómez-Brandón, A. Hogan, A. Irribarra-Cortés, G. Navarro, and J. Reutter, “Worst-case-optimal joins on graphs with topological relations,” in *Proceedings of the ACM on Web Conference 2025*, ser. WWW ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 59–71.
- [9] G. Jin, X. Feng, Z. Chen, C. Liu, and S. Salihoglu, “Kùzu graph database management system,” in *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8–11, 2023*. www.cidrdb.org, 2023.
- [10] D. Vrgoc, C. Rojas, R. Angles, M. Arenas, V. Calisto, B. Farias, S. Ferrada, T. Heuer, A. Hogan, G. Navarro, A. Pinto, J. L. Reutter, H. Rosales-Méndez, and E. Toussaint, “Millenniumdb: A multi-modal, multi-model graph database,” in *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9–15, 2024*, P. Barceló, N. Sánchez-Pi, A. Meliou, and S. Sudarshan, Eds. ACM, 2024, pp. 496–499.
- [11] T. Neumann and M. J. Freitag, “Umbra: A disk-based system with in-memory performance,” in *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12–15, 2020, Online Proceedings*. www.cidrdb.org, 2020.
- [12] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, “Design and implementation of the logicblox system,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 1371–1382.
- [13] A. Hogan, C. Riveros, C. Rojas, and A. Soto, “A worst-case optimal join algorithm for sparql,” in *The Semantic Web - ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2019, pp. 258–275.
- [14] A. Birler, A. Kemper, and T. Neumann, “Robust join processing with diamond hardened joins,” *Proc. VLDB Endow.*, vol. 17, no. 11, pp. 3215–3228, 2024.
- [15] J. Wu and D. Suciu, “Honeycomb: A parallel worst-case optimal join on multicores,” *Proc. ACM Manag. Data*, vol. 3, no. 3, Jun. 2025.
- [16] D. Arroyuelo, A. Gómez-Brandón, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto, “The ring: Worst-case optimal joins in graph databases using (almost) no extra space,” *ACM Trans. Database Syst.*, vol. 49, no. 2, Mar. 2024.
- [17] J. Wang, I. Trummer, A. Kara, and D. Olteanu, “Adopt: Adaptively optimizing attribute orders for worst-case optimal join algorithms via reinforcement learning,” *Proc. VLDB Endow.*, vol. 16, no. 11, p. 2805–2817, Jul. 2023.
- [18] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica, “Learning to optimize join queries with deep reinforcement learning,” *arXiv preprint arXiv:1808.03196*, 2018.
- [19] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, “Neo: a learned query optimizer,” *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1705–1718, Jul. 2019.
- [20] V. Markl, “Making learned query optimization practical: A technical perspective,” *SIGMOD Rec.*, vol. 51, no. 1, p. 5, Jun. 2022.
- [21] X. Yu, G. Li, C. Chai, and N. Tang, “Reinforcement learning with tree-lstm for join order selection,” in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1297–1308.
- [22] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, “Adopting worst-case optimal joins in relational database systems,” *Proc. VLDB Endow.*, vol. 13, no. 12, p. 1891–1904, Jul. 2020.
- [23] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré, “Emptyheaded: A relational engine for graph processing,” in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 431–446.
- [24] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoc, “Foundations of modern query languages for graph databases,” *ACM Comput. Surv.*, vol. 50, no. 5, Sep. 2017.
- [25] A. Atserias, M. Grohe, and D. Marx, “Size bounds and query plans for relational joins,” *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1737–1767, 2013.
- [26] B. Hilprecht, C. Binnig, and U. Röhm, “Towards learning a partitioning advisor with deep reinforcement learning,” in *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, ser. aiDM ’19. New York, NY, USA: Association for Computing Machinery, 2019.
- [27] G. Li, X. Zhou, S. Li, and B. Gao, “Qtune: a query-aware database tuning system with deep reinforcement learning,” *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2118–2130, Aug. 2019.
- [28] D. Van Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Bilien, and A. Pavlo, “An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems,” *Proc. VLDB Endow.*, vol. 14, no. 7, p. 1241–1253, Mar. 2021.
- [29] J. Wang, I. Trummer, and D. Basu, “Demonstrating udo: A unified approach for optimizing transaction code, physical design, and system parameters via reinforcement learning,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2794–2797.
- [30] I. Trummer, “Db-bert: A database tuning tool that “reads the manual,”” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 190–203.
- [31] C. Fan, Z. Pan, W. Sun, C. Yang, and W.-N. Chen, “Latuner: An llm-enhanced database tuning system based on adaptive surrogate model,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2024, pp. 372–388.
- [32] W. Sun, Z. Pan, Z. Hu, Y. Liu, C. Yang, R. Zhang, and X. Zhou, “Rabbit: Retrieval-augmented generation enables better automatic database knob tuning,” in *ICDE*, 2025, pp. 3807–3820.
- [33] H. Wang, Y. Zhang, L. Qin, W. Wang, W. Zhang, and X. Lin, “Reinforcement learning based query vertex ordering model for subgraph matching,” in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9–12, 2022*. IEEE, 2022, pp. 245–258.
- [34] Z. Wu, R. Marcus, Z. Liu, P. Negi, V. Nathan, P. Pfeil, G. Saxena, M. Rahman, B. Narayanaswamy, and T. Kraska, “Stage: Query execution time prediction in amazon redshift,” in *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9–15, 2024*, P. Barceló, N. Sánchez-Pi, A. Meliou, and S. Sudarshan, Eds. ACM, 2024, pp. 280–294. [Online]. Available: <https://doi.org/10.1145/3626246.3653391>
- [35] Z. Pan, Y. Zhang, C. Yang, A. Ghazal, R. Zhang, H. Hu, X. Wu, Y. Dong, and X. Zhou, “Hyper: Hybrid physical design advisor with multi-agent reinforcement learning,” in *ICDE*, 2025, pp. 1565–1578.
- [36] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19–24, 2016*, ser. JMLR Workshop and Conference Proceedings,

- M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 1928–1937.
- [37] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016.
 - [38] A. Kamali, V. Kantere, C. Zuzarte, and V. Corvinelli, “Robust plan evaluation based on approximate probabilistic machine learning,” *Proc. VLDB Endow.*, vol. 18, no. 8, pp. 2626–2638, 2025. [Online]. Available: <https://www.vldb.org/pvldb/vol18/p2626-kamali.pdf>
 - [39] Z. Pan, Y. Wang, Y. Zhang, S. B. Yang, Y. Cheng, P. Chen, C. Guo, Q. Wen, X. Tian, Y. Dou *et al.*, “Magicscaler: Uncertainty-aware, predictive autoscaling,” *PVLDB*, vol. 16, no. 12, pp. 3808–3821, 2023.
 - [40] “Apex source code and supplementary material,” <https://github.com/MKMaS-GUET/APEX>.
 - [41] R. Angles, C. Buil-Aranda, A. Hogan, C. Rojas, and D. Vrgoc, “Wd-bench: A wikidata graph query benchmark,” in *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, U. Sattler, A. Hogan, C. M. Keet, V. Presutti, J. P. A. Almeida, H. Takeda, P. Monnin, G. Pirrò, and C. d’Amato, Eds., vol. 13489. Springer, 2022, pp. 714–731.
 - [42] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, “Diversified stress testing of rdf data management systems,” in *The Semantic Web - ISWC 2014: 13th International Semantic Web Conference, Riva Del Garda, Italy, October 19-23, 2014. Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 197–212.
 - [43] P. Trivedi, G. Maheshwari, M. Dubey, and J. Lehmann, “Lc-quad: A corpus for complex question answering over knowledge graphs,” in *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, C. d’Amato, M. Fernández, V. Tamma, F. Lécué, P. Cudré-Mauroux, J. F. Sequeda, C. Lange, and J. Hefflin, Eds., vol. 10588. Springer, 2017, pp. 210–218.
 - [44] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum, “YAGO2: A spatially and temporally enhanced knowledge base from wikipedia,” *Artif. Intell.*, vol. 194, pp. 28–61, 2013.
 - [45] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis, “A survey and experimental comparison of distributed SPARQL engines for very large RDF data,” *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2049–2060, 2017.
 - [46] R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, “Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning,” *VLDB J.*, vol. 25, no. 3, pp. 355–380, 2016.
 - [47] D. Justen, D. Ritter, C. Fraser, A. Lamb, N. Tran, A. Lee, T. Bodner, M. Y. Haddad, S. Zeuch, V. Markl, and M. Boehm, “POLAR: adaptive and non-invasive join order selection via plans of least resistance,” *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1350–1363, 2024.
 - [48] T. Chen, J. Gao, Y. Tu, and M. Xu, “GLO: towards generalized learned query optimization,” in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2024, pp. 4843–4855.