DQN-Guided Rule-Class Selection for Search-Based Query Rewriting

by

Shengchen Liu

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the
Master of Computer Science degree
Concentration in Applied Artificial Intelligence

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Query rewrite transforms a SQL query into a semantically equivalent form expected to execute more efficiently. Search-based rewriters such as LearnedRewrite organize alternative rewrite orders as a policy tree and explore it with Monte Carlo Tree Search. In the publicly released LearnedRewrite implementation, expansion enumerates every applicable rule class at each search node, and each candidate triggers a rewrite execution and a PostgreSQL EXPLAIN call, making expansion a measurable contributor to rewrite latency.

This thesis formulates expansion-time rule-class selection as a state-dependent decision problem and proposes DQRS (DQN-Guided Rule-Class Selection), which ranks the legal rule classes at each state with a Deep Q-Network and forwards only the top k to the unchanged rewrite routine. On an in-domain workload derived from the TPC-H benchmark at $k = 1$, average rewrite latency falls from 15.3 ms to 4.0 ms and average policy-tree size from 9.9 to 2.4 nodes, reductions of about 74% and 76%. Average cost reduction is preserved while the median falls from 156 to 103, so the method exposes an explicit operating point on the efficiency–quality trade-off rather than dominating the baseline. Under an identical legal-action mask and at the same node count, learned ranking attains a median cost reduction of 103 against 13 for random selection. Transferred without retraining to a workload derived from the TPC-DS benchmark at $k = 4$, DQRS reduces latency by about 24% and node count by about 30% at unchanged cost reduction, but in this saturated regime it is not distinguishable from random selection.

The baseline throughout is the released implementation, which omits the learned cost estimator of the original paper, and all quality figures are optimizer-side estimates rather than measured execution time. Extending the evaluation from optimizer estimates to end-to-end execution time is the primary direction for future work.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Verena Kantere, for her guidance, support, and patience throughout my master's study. Her knowledge and experience in database systems and query optimization have been invaluable to the development of this thesis. Her constructive feedback helped me refine the direction of my research, improve the clarity of my ideas, and approach problems with greater rigor. I am also grateful for her encouragement during the difficult stages of this work, which gave me the confidence to continue when the research process became challenging.

I would also like to thank my parents for their unconditional love, trust, and support. Their encouragement has been a constant source of strength throughout my studies abroad. Their belief in me has helped me overcome difficulties and remain focused on completing this degree.

Finally, I am thankful to everyone who supported me during my time at the University of Ottawa. This thesis would not have been possible without their guidance, understanding, and encouragement.

Table of Contents

List of Tables

List of Figures

List of Acronyms

CPU	Central Processing Unit
CSV	Comma-Separated Values
DBMS	Database Management System
DQN	Deep Q-Network
DQRS	DQN-Guided Rule-Class Selection
GPU	Graphics Processing Unit
JSON	JavaScript Object Notation
LLM	Large Language Model
MCTS	Monte Carlo Tree Search
ONNX	Open Neural Network Exchange
RL	Reinforcement Learning
SQL	Structured Query Language
TPC	Transaction Processing Performance Council
TPC-DS	TPC Decision Support benchmark
TPC-H	TPC ad-hoc decision support benchmark
UCT	Upper Confidence Bound applied to Trees

Chapter 1

Introduction

In modern database management systems, queries written in structured query language (SQL) are routinely transformed before they are executed. A query that produces the correct result is not necessarily a query that executes efficiently, because the same result can be obtained from many semantically equivalent SQL forms whose execution costs differ widely. Bridging the gap between a user-written query and an efficient one is part of the responsibility of the database management system, and is performed during query processing through a sequence of internal transformations. Among these transformations, query rewrite operates at the logical level: it replaces a given query with a semantically equivalent query that is expected to be more efficient before any physical execution plan is chosen. The quality of these logical-level transformations affects every downstream stage of query execution, which makes query rewrite an important subject of study in both database systems research and practical database deployments. This thesis investigates query rewrite, with a focus on how a learned value model can guide the search through the space of semantically equivalent rewrites within an existing rewrite framework.

1.1 Query Rewrite and Query Optimization

Modern data-intensive applications rely on database management systems (DBMSs) to store and retrieve large volumes of structured data. Users interact with a DBMS by submitting queries, typically expressed in SQL, that describe what information is needed without specifying how the data should be retrieved. The DBMS is then responsible for translating each query into an executable plan and producing the result. As datasets grow and queries become more complex, the time taken to evaluate a query increasingly affects system responsiveness and user experience.

A given SQL query can usually be evaluated in many different ways. Even when two formulations or two execution plans return identical results, the resources they consume, including CPU time, memory, disk I/O, and the volume of intermediate data, can differ by orders of magnitude [20]. This variability comes from the declarative nature of SQL: it specifies the desired output rather than the procedure used to obtain it. The DBMS

therefore has substantial latitude in how a query is evaluated, and that choice has a direct impact on execution efficiency.

Choosing an efficient evaluation strategy is known as query optimization [7, 17]. Query optimization operates at two conceptually distinct layers. At the logical layer, the input query is transformed into another query that is semantically equivalent but may expose more favorable structural properties [17]. Examples of such transformations include converting a subquery into a join, removing redundant aggregates, and simplifying predicates [17]. At the physical layer, the optimizer selects concrete execution strategies for the resulting query, such as access paths, join algorithms, and operator implementations. The two layers interact: the logical formulation passed to the physical optimizer constrains which physical plans are available, so transformations at the logical layer can affect which optimization opportunities remain visible downstream.

Logical-layer transformations are realized through query rewrite. Query rewrite changes the formulation of the input query while preserving its semantics, with the goal of producing a form that is more efficient to execute. Even when the underlying physical optimizer is highly capable, its effectiveness is shaped by the logical form of the query it receives. Query rewrite is therefore widely regarded as an important component of query optimization in modern DBMSs [2, 17, 30].

This thesis is concerned with query rewrite as a logical-layer activity. The remainder of the chapter introduces the basic concepts that recur throughout the thesis (Section 1.1.1) and shows how they compose into the three-layer structure the thesis is organized around (Section 1.1.2), describes the query rewrite task more precisely (Section 1.2), motivates the specific problem addressed in this thesis (Section 1.3), and outlines the methodology and contributions (Sections 1.4 and 1.5).

1.1.1 Concepts

Query rewrite involves a set of recurring concepts: about queries themselves, the rules that transform them, and the structure over which transformations are organized. This section gathers definitions of the terms that recur throughout the thesis and fixes their meaning in the present context. The definitions are kept compact; mechanisms and algorithms associated with these concepts are introduced where they become relevant in later chapters.

- **Query:** A query is a structured request, typically expressed in SQL, for the DBMS to retrieve, transform, or otherwise act upon data stored in one or more relations. Throughout this thesis, the term refers specifically to read-oriented SELECT queries, since these are the primary target of query rewrite.
- **Logical Plan:** A logical plan is a tree-structured representation of the relational operations implied by a query. Internal nodes correspond to relational operators (such as join, filter, project, and aggregate) and leaves correspond to base relations. The logical plan abstracts away physical execution details and provides the substrate over which query rewrite operates.

- **Rewrite Rule:** A rewrite rule describes a structural transformation that maps a fragment of one logical plan to a semantically equivalent fragment with a different shape. A rule typically consists of three parts: a pattern that specifies which fragments are eligible, a set of validity conditions, and a transformation that produces the rewritten fragment. Different systems formalize rules differently, but the underlying structure is broadly consistent.

- **Rule Class:** A rule class is a category of rewrite rules that target the same family of operators or transformation patterns. For example, all rules that operate on filter predicates, or all rules that operate on join structures, are examples of rule classes. Rule classes are the level at which expansion is organized in the implementation considered in this thesis.

- **Equivalence:** Two queries are considered equivalent if they return the same result on every valid input database. In practice, equivalence is the correctness condition that any rewrite must preserve: a transformation that changes performance but alters output semantics is not a valid rewrite. Whether a particular rule preserves equivalence may depend on assumptions about the schema, on the absence of duplicates, or on the presence of NULL values, so equivalence is generally checked under stated conditions instead of being treated as universal.

- **Cost:** Cost is a numerical estimate of how expensive a query is to execute, used to compare alternatives during optimization. Throughout this thesis, cost values are obtained from the underlying DBMS optimizer, specifically from the PostgreSQL EXPLAIN output, not from direct measurement of execution time. Cost is therefore an optimizer-side proxy for execution effort: useful for ranking alternative plans within the same system, but not a guarantee of true runtime. The cost reduction achieved by a rewrite is the difference between the cost estimated for the original query and the cost estimated for the rewritten query. How the optimizer produces these estimates, and how they are retrieved from PostgreSQL, are described in Section 2.1.2.

- **Search Space:** The search space of query rewrite is the collection of all rewrite sequences that can be produced from an input query under a given rule set. Because rules can be applied in many orders, and because applying one rule may enable further rules, the size of this space grows rapidly with query complexity. Enumerating it exhaustively is generally infeasible within practical rewrite latency, so a search procedure must focus effort on a small subset of promising sequences rather than evaluate all of them.

- **Policy Tree:** A policy tree [30] is a rooted tree used to organize the search space of rewrite sequences. The root corresponds to the original query, each non-root node corresponds to a rewritten state obtained from its parent by one rewrite step, and each path from the root to a node corresponds to one concrete rewrite order. Different rewrite orders that share an initial prefix share the corresponding upper-level nodes in the tree. The policy tree gives the search procedure an explicit structure to operate over rather than leaving rule order implicit in a fixed traversal.

- **Rewrite State**: A rewrite state is the intermediate result of applying zero or more rewrite rules to the input query. It is held internally as a relational tree, the same structure described above as a logical plan, and it is what a node of the policy tree represents. The initial state is the input query itself. Because every decision made during rewrite is made at some state, the term is used throughout this thesis wherever the choice of the next rule depends on what has been rewritten so far.

- **Expansion**: Expansion is the step in which a search procedure generates the children of a policy-tree node. Given a node and the rule classes legally applicable at its state, expansion attempts each selected class in turn, and each attempt performs one rewrite execution and one cost evaluation. Nodes that pass the executability and cost-acceptance conditions are added to the tree as children. Expansion is the point at which this thesis intervenes; the mechanism is described in Section 3.3.3.

- **Two Senses of Efficiency**: The word "efficient" is used in the query rewrite literature, and in this thesis, in two distinct senses, and keeping them apart is necessary to state what this thesis does and does not claim. An *efficient query* is one whose evaluation is cheap: it has a low estimated or measured execution cost once the DBMS runs it. Producing an efficient query is the *goal* of query rewrite. An *efficient rewrite procedure* is one that reaches its output cheaply: it spends little time searching, generating candidates, and evaluating them before returning a rewritten query. This is the *cost of pursuing* that goal, measured in this thesis as rewrite latency (Section 4.1.4). The two are independent. A rewrite procedure that explores exhaustively may return a highly efficient query while itself being slow, and a procedure that returns almost immediately may return a query no better than the input. Separately from both, an *approach* may be described as *simple*, which is a statement about its conceptual and implementational complexity, not about the queries it produces: a fixed-order heuristic rewriter is simple in this sense regardless of how good or bad its output turns out to be. Throughout this thesis, the goal of query rewrite is stated in the first sense, whereas the contribution of the proposed method is stated in the second: DQRS aims to make the rewrite *procedure* cheaper while limiting degradation in the quality of the query it *produces*. Where the intended sense could be ambiguous, the qualified forms "execution cost" and "rewrite-time overhead" are used in preference to the bare word "efficient".

1.1.2 How the Three Layers Fit Together

The definitions above operate at three different levels, and the relationship among them is what the rest of this thesis is organized around. Because these three levels recur in every subsequent chapter, they are stated together here before any of them is developed in detail.

Query rewrite is the task. The object of interest is a semantics-preserving transformation from an input query q_0 to an output query q_{out} that is expected to execute more cheaply. As Section 1.2 develops, this transformation is not a single step but a sequence of local steps $q_0 \to q_1 \to \cdots \to q_{\text{out}}$, and because rules interact, the order of those steps

changes the outcome. The open question at this level is therefore not which rules exist but which step to take next.

Rule classes are the unit in which that step is chosen. The framework used in this thesis does not expose individual rewrite rules to the search procedure. Candidate generation is organized around seven rule classes, each covering a family of rules that target the same operators, and together covering 57 concrete Apache Calcite rule instances (Section 3.2.2, Table 3.1). At any given rewrite state only some of the seven are legally applicable, since a class whose operator pattern is absent from the current relational tree cannot fire. Deciding what to do next at a given state therefore reduces to a concrete and finite question: which of the currently legal rule classes should be attempted, given that each attempt costs a rewrite execution and a cost evaluation.

Deep reinforcement learning supplies the answer to that question. A rule class cannot be judged by its immediate effect alone, because applying it changes which rewrites become reachable afterwards; the choice is thus sequential rather than one-shot. This is the standard setting for reinforcement learning, and it maps onto the present problem directly: the current rewrite state is the state, the seven rule classes are the actions, and the change in estimated query cost across one rewrite step is the reward. A Deep Q-Network trained offline on transitions recorded from the existing search procedure estimates a value for each action at a given state; at rewrite time these values are masked to the legal classes, ranked, and truncated to the top-k, and only those k classes are passed to the unchanged rewrite routine.

In short, query rewrite defines the problem, rule classes define the action space, and the Deep Q-Network provides a ranking over that action space. The learned component does not generate SQL, does not decide which node of the search to visit, and does not alter the rules themselves; it only orders a set of at most seven candidate actions at each expansion step. Figure 1.1 summarizes this relationship.

1.2 The Query Rewrite Task

Although query rewrite is sometimes described as if it were a single transformation, in practice it is more accurately understood as a sequence of local transformations applied to a query or its logical plan. Each individual transformation matches a fragment of the current state and produces a modified state. Repeating this process over multiple steps yields a chain of rewritten queries, ending when no further applicable rule remains or when a rewrite budget runs out. The query that is finally chosen for execution is the result of this multi-step process rather than of any single rule application.

Once query rewrite is viewed as a sequence of local transformations, the order in which rules are applied becomes consequential. Rules can interact: applying one rule may enable, disable, or alter the usefulness of subsequent rules [7, 30]. Two rule sequences that contain the same rules in different orders may therefore lead to different intermediate states, different downstream rewrite opportunities, and ultimately different final queries. Even when

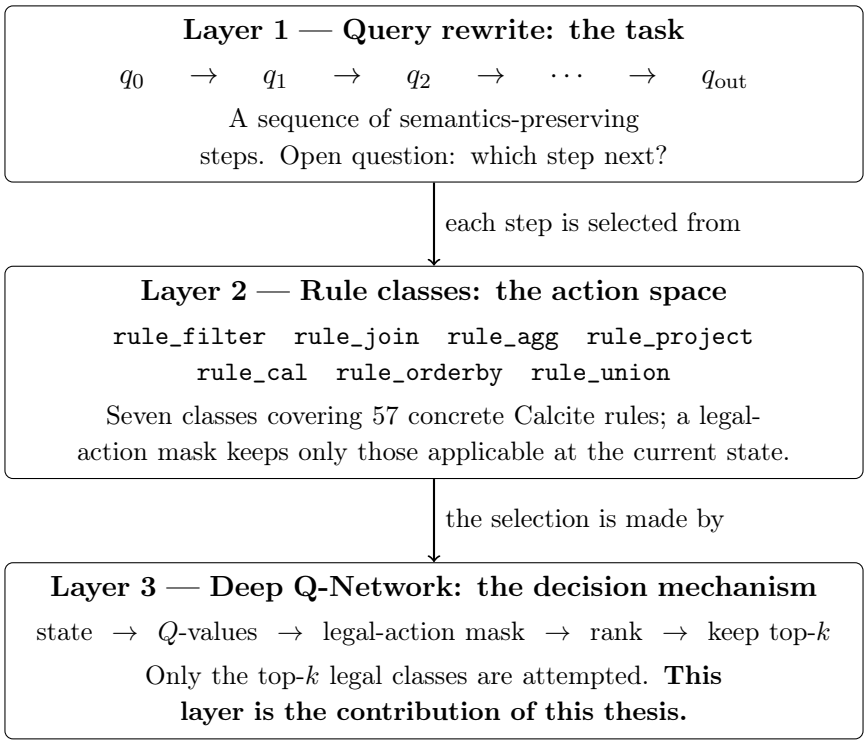

Figure 1.1: The three layers this thesis operates across. Query rewrite defines the problem, rule classes define the action space, and the Deep Q-Network provides a ranking over that action space. The surrounding search procedure, the rewrite rules themselves, and the executability and cost-acceptance checks are unchanged. The seven rule classes and the 57 concrete Calcite rule instances they cover are listed in Section 3.2.2 and Table 3.1.

each individual rule is well understood, navigating the resulting space of rewrite sequences is non-trivial.

A natural question is why rewrite proceeds as a sequence of local steps rather than as a single transformation from the input query to a final rewritten form. The answer follows from what a rewrite rule is. Each rule is defined over a fragment of a logical plan: it matches a specific structural pattern and replaces it with an equivalent one under stated validity conditions. Equivalence is established for the fragment the rule rewrites, not for arbitrary pairs of queries. There is consequently no rule that maps an arbitrary input query directly to a preferred final form, and such a mapping exists only as a composition of individual rule applications. An approach that produced the final query in one step, for instance by generating it with a language model (Section 2.5.3), would give up the property that makes rule-based rewrite dependable: the equivalence of the output would have to be verified after the fact rather than guaranteed by construction. The step-by-step formulation is therefore not a design preference but a consequence of building rewrite from verified local transformations, and it is what makes the order of those steps a decision problem in the first place.

To make this concrete, consider a simple example. Suppose the query in question is:

```
SELECT t1.*
FROM t1
WHERE a1 IN (
    SELECT a2 FROM t2 WHERE a2 < 2
);
```

One possible rewrite transforms the IN-subquery into an explicit join:

```
SELECT DISTINCT t1.*
FROM t1
JOIN t2 ON t1.a1 = t2.a2
WHERE t2.a2 < 2;
```

Both formulations are intended to return the same set of rows, but the rewritten form exposes a different structure that the underlying optimizer may handle more effectively. This particular rewrite is only valid under specific equivalence conditions, and whether it is beneficial depends on data statistics, available indexes, and the optimizer's downstream choices. Other common rewrite transformations include simplifying redundant predicates, converting outer joins into inner joins under preserving conditions [6], eliminating redundant aggregates, and merging or splitting subqueries [17]. Each of these may be appropriate or inappropriate depending on the surrounding query state.

Three properties make this space difficult to navigate, and they recur throughout the thesis. First, it grows combinatorially: the number of distinct rewrite orders increases with both the size of the query and the number of applicable rules, so exhaustive enumeration is not available within any practical rewrite budget. Second, applicability does not imply usefulness. Pattern matching establishes only that a rule can be applied at the current state; it carries no information about whether applying it will reduce cost, or whether it

will lead to a state from which further reductions become available. Third, evaluating a candidate is not free. Determining whether a rule application is beneficial requires executing the rewrite and obtaining a cost estimate for the resulting state, which in the framework used in this thesis means a call to the underlying DBMS optimizer. The cost of deciding what to rewrite is therefore itself part of the cost of rewriting, and it is this cost that the present work targets.

Once the existence of multiple rewrite alternatives is acknowledged, the central question is no longer whether useful rules exist but how to navigate the space of possible rewrite sequences efficiently and effectively. This question motivates the present thesis. The next section discusses how this navigation problem has been approached in prior work, identifies specific limitations of current approaches, and frames the problem addressed in this thesis.

1.3 Motivation and Problem Statement

The simplest approach to query rewrite is to apply rules under a fixed traversal order, such as top-down, bottom-up, or expert-defined orderings, as exemplified by traditional rule engines in production database systems [2, 18]. These fixed-order strategies are easy to implement and cheap to run at rewrite time. They also have a clear weakness: the ordering is fixed at design time, so it cannot adapt to the rule interactions present in a particular query, and a locally reasonable choice is not always the best choice for the overall rewrite outcome. Fixed-order heuristic rewrite may therefore miss better rewrite sequences in queries where rule interactions are non-trivial.

The general idea of applying transformation rules within a structured search over equivalent plans has a long history in cost-based query optimization [7, 8]. LearnedRewrite [30] applies this perspective specifically to logical-layer query rewrite: alternative rewrite orders are organized as a policy tree, and the search is guided by Monte Carlo Tree Search [3, 10] and a learned cost estimator. LearnedRewrite serves as the methodological starting point of this thesis.

One detail should be made explicit before going further, since it affects how the rest of this thesis is framed. The system used as the practical reference here is the publicly released single-rule implementation of LearnedRewrite [29] (hereafter referred to as the *released implementation*), not the system proposed in the paper in its full form. As verified by inspection of the publicly released codebase and discussed in detail in Section 3.1, the released implementation realizes the policy-tree search structure but does not expose the learned cost estimator described in the paper; cost reduction is instead obtained from the underlying DBMS optimizer through PostgreSQL EXPLAIN.

The three limitations below all concern the released implementation just described, not the system proposed in the paper; they stand out as practically important for the efficiency of rewrite search itself.

- **Expansion is a non-trivial and concrete component of rewrite latency.** Within the released search loop, expansion proceeds by enumerating all rule classes

applicable to the current query state, and each candidate triggers concrete rewrite execution, successor-state generation, and cost evaluation through PostgreSQL EXPLAIN. As search proceeds and the policy tree grows, the cumulative cost of these per-candidate operations is therefore not abstract: it contributes directly to measured rewrite latency. This makes expansion a meaningful target for efficiency improvement, even before claiming it is the dominant source of overhead. Because every node in the policy tree is produced by one such cycle of rewrite execution and cost evaluation, the number of nodes generated during a rewrite is a direct structural proxy for this overhead. Reducing the size of the policy tree is therefore not a preference for compact search structures in the abstract; it corresponds to performing fewer rewrite executions and fewer calls to the optimizer. Policy-tree size is reported alongside rewrite latency throughout Chapter 4 for this reason.

- **Existing pruning operates at coarse, query-agnostic granularity rather than as a state-dependent decision.** While the released implementation includes basic gating mechanisms based on rule applicability and elementary cost-based filtering, these mechanisms do not adapt their preferences to the current rewrite state. The same rule class is treated similarly regardless of whether the current state contains operator patterns that are likely to benefit from it. Expansion candidates that are unlikely to be useful at the current state still pay the full cost of generation and evaluation.

- **Local immediate-gain signals do not adequately reflect downstream rewrite value.** The value of a rewrite action is determined not only by its immediate effect but also by how it shapes the rewrite states that become reachable afterward. A rewrite with limited immediate gain may still lead to a state where substantially more beneficial rewrites become available, while a rewrite that looks locally favorable may dead-end. Pruning strategies based purely on immediate cost reduction cannot distinguish these cases, so reducing expansion effectively requires a signal that goes beyond per-candidate immediate gain.

Of the three limitations above, the first establishes expansion as a meaningful intervention point. The second describes a property of the released implementation that this thesis directly addresses: existing pruning is state-agnostic, and a state-dependent decision mechanism can replace it. The third describes a deeper property of the released implementation: the released code does not include a signal that captures downstream rewrite value. This property originates in the omission of the paper's learned cost estimator from the released code, as discussed in Section 3.1.2. Fully addressing the third limitation would require restoring or reproducing that estimator, which is outside the scope of this thesis.

The problem studied here is therefore narrower than the union of all three limitations identified above. More concretely, consider a selected policy-tree node v within the search loop of the released implementation. Let $s(v)$ denote the rewrite state reached at v, represented internally as a relational tree, and let $\mathcal{A}(v)$ denote the rule classes legally applicable at that state, drawn from the fixed set of seven rule classes exposed by the released implementation (enumerated in Section 3.2.2). For a configured top-k value k, the

expansion-time selection problem addressed in this thesis is to choose a subset $S_k(v) \subseteq \mathcal{A}(v)$ with $|S_k(v)| \leq k$, whose rule classes are forwarded to the existing single-rule rewrite routine for rewrite attempts; whether a given attempt yields an accepted child is determined by the unchanged executability and cost-acceptance checks of the released implementation (Algorithm 1). The intervention is therefore explicitly not aimed at generating final SQL directly, replacing Monte Carlo Tree Search (MCTS), reconstructing the learned cost estimator proposed in the LearnedRewrite paper, or modifying the candidate-generation interface below the rule-class level. The objective is to reduce rewrite-search overhead by avoiding low-value rule-class attempts while limiting degradation in the released implementation's PostgreSQL EXPLAIN-based cost-reduction signal, which is also the quality signal used internally by the released implementation. This formulation deliberately frames the intervention as exposing an explicit operating point on an efficiency–quality trade-off rather than as Pareto-improving on the released baseline; the location of this operating point on the in-domain workload is reported in Chapter 4.

The third limitation is therefore addressed only partially: the value model used by the proposed method (Section 3.3) can in principle propagate value information across recorded rewrite transitions, although the strength of this propagation under the released search budget is constrained by short recorded trajectories (Section 3.3.4). A full treatment of the third limitation would require reproducing the paper's selection-stage cost estimator, which is outside the scope of this thesis.

Problem Statement.

Given:

- the search loop of the released LearnedRewrite single-rule implementation, which maintains a policy tree and selects nodes for expansion under a fixed rewrite budget;
- the fixed set $\mathcal{R}$ of seven rule classes exposed by the released implementation (Section 3.2.2);
- at each selected policy-tree node v, the current rewrite state $s(v)$, represented internally as a relational tree, and the legal-action set $A(v) \subseteq \mathcal{R}$ of rule classes applicable at $s(v)$;
- a configured selection budget $k \geq 1$.

Find: for each selected node v, a subset $S_k(v) \subseteq A(v)$ with $|S_k(v)| \leq k$, whose rule classes are forwarded to the existing single-rule rewrite routine for expansion.

Subject to: the surrounding search loop, the candidate-generation interface below the rule-class level, the PostgreSQL EXPLAIN-based executability check, and the cost-acceptance check of the released implementation (Algorithm 1) remain unchanged.

Objective: reduce rewrite-search overhead, measured by end-to-end rewrite latency and policy-tree size, while limiting degradation in the workload's PostgreSQL EXPLAIN-based cost-reduction signal relative to the released implementation operating with the exhaustive choice $S_{|A(v)|}(v) = A(v)$ at every v.

Out of scope: generating final SQL directly, replacing MCTS, reconstructing the learned cost estimator described in the LearnedRewrite paper, and modifying the candidate-generation interface below the rule-class level.

1.4 Methodology

The research presented in this thesis is conducted in several phases. The first is a literature review of search-based query rewrite, including LearnedRewrite and other learning-guided approaches, which establishes the methodological context and identifies remaining gaps relevant to this work. The second is an analysis of the released implementation, focused on the expansion stage where the search overhead is concretely incurred. The third formalizes expansion-time rule-class selection as a state-dependent decision problem, defining the relevant state representation, action space, reward signal, and value model. The fourth implements the proposed method as a module-level modification of the released implementation, leaving the surrounding policy-tree search structure unchanged. The fifth evaluates the method experimentally on a held-out test split of a release-aligned TPC-H-derived workload [25] and on a separately constructed TPC-DS-derived validation workload [24].

The proposed method follows an offline training and online inference workflow. Training data are collected by instrumenting the original search process and recording rewrite transitions. The value model is then trained offline in a separate environment, using only the recorded transitions as supervision. After training, the model is exported and integrated into the rewrite framework as an inference-only component during online query rewriting. No online policy update or further training occurs during evaluation. This separation keeps the rewrite framework independent from the training environment while allowing the trained model to guide expansion decisions through lightweight inference.

The methodology is intentionally scoped: the thesis studies a focused intervention inside an existing search-based rewrite framework rather than proposing a new framework or replacing the surrounding rewrite system. Subsequent chapters describe each phase of this methodology in detail.

1.5 Contributions

To address the limitations identified in Section 1.3, this thesis proposes DQRS (DQN-guided Rule-class Selection), a method for guiding expansion-time rule-class selection within the existing search-based query rewrite framework. DQRS treats rule-class selection during expansion as a state-dependent decision problem, replaces exhaustive enumeration of legal rule classes with learned ranking and top-k selective expansion, and is integrated into the released implementation as a module-level modification rather than as a redesign of the surrounding rewrite system. The main contributions of this thesis are summarized as follows.

- **A scoped problem formulation that targets expansion-time rule-class selection as a distinct intervention point.** This thesis formulates expansion-time rule-class selection as a state-dependent decision problem within search-based query rewrite. The starting observation is that expansion in the released implementation is not a free branching step: each candidate rule class triggers concrete rewrite execution and cost evaluation, so the cost of generating candidates contributes directly to overall rewrite latency. Building on this observation, expansion-time rule-class selection is formulated as a state-dependent decision problem: at each search state, the system must choose which rule classes to expand from a finite, state-dependent set of legal alternatives. This formulation isolates a narrow yet practically important decision point inside the existing rewrite framework, making it possible to study a focused intervention without redesigning the surrounding search procedure.

- **A state-dependent value-guided method for selective expansion: DQRS.** The proposed method implements rule-class selection as a state-dependent ranking problem. The current rewrite state is encoded as a fixed-length feature vector capturing structural properties of the relational tree and coarse cost-related signals. A Q-network [16] trained offline over rewrite transitions estimates action values for the rule classes legal at the current state, and a top-k subset of the highest-ranked legal rule classes is forwarded to the existing single-rule rewrite routine for expansion. Legal-action masking, value-based ranking, and bounded top-k together let the method be selective without collapsing to a single greedy choice. The method directly addresses the state-agnostic nature of existing pruning identified in Section 1.3. The bootstrap term in the training target propagates value information across recorded transitions whose next state has not been empirically confirmed to be terminal, providing a learned ordering signal that goes beyond per-transition reward at a single state.

- **An empirical characterization of when state-dependent learned rule-class selection produces net efficiency gains, and where it does not.** The released implementation is used as the experimental substrate throughout. Reported gains are relative to the released implementation rather than to the full framework proposed in the paper, and rewrite quality is measured by PostgreSQL EXPLAIN cost, the same estimator the released implementation uses internally. On the held-out test split of a release-aligned TPC-H-derived workload, the empirical operating range over which DQRS Pareto-improves on the released baseline is narrow. At $k = 1$, rewrite latency and search-space size drop substantially while the workload-level average cost reduction is preserved, but the typical-case (median) cost reduction is reduced. At $k = 2$, the median is restored but the latency advantage is lost. The detailed numbers are reported in Chapter 4. Within this trade-off, the value of learned ranking over unlearned pruning is supported by the random-k comparison at $k = 1$ on the same workload, where random selection under the same legal-action mask produces a substantially lower median cost reduction than DQN-guided ranking. On the TPC-DS-derived out-of-domain validation workload, DQRS at $k = 4$ reduces rewrite latency and search effort relative to the released baseline while preserving aggregate

cost reduction; however, random selection at the same k matches the released baseline on cost reduction, and DQRS at $k = 5$ yields no further cost-reduction gain over $k = 4$ despite increased exploration, indicating that the comparison at $k = 4$ occurs near a workload-level cost-reduction ceiling. The present experiments therefore do not establish that the learned ranker transfers across workloads. Ablation studies further isolate the effect of the top-k pruning level and the gap between learned and unlearned selection under matched masking.

Together, these contributions support a scoped claim: within the released LearnedRewrite single-rule codebase, a state-dependent learned intervention at the expansion stage produces a measurable efficiency gain at a specific operating point on the in-domain TPC-H-derived workload, in exchange for a measurable typical-case quality regression at that point; at less aggressive operating points on the same workload, the intervention does not improve over the released baseline. On the cross-workload TPC-DS-derived setting, the present experiments do not establish that the learned preferences transfer beyond the training distribution.

Two scope conditions shape these claims: (1) the released-baseline framing is comparative against the released code, not against a fully reproduced version of the system proposed in the paper (Section 3.1.3), and (2) the empirical study has multiple boundaries, including workload construction, the use of estimator-side cost as the quality signal, and offline reinforcement learning concerns, each discussed in Section 4.5.

1.6 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents the background needed to understand the thesis, covering query rewrite fundamentals, rule-based rewrite mechanisms, policy-tree representation, Monte Carlo Tree Search, the deep reinforcement learning concepts relevant to value-based action selection, and a comparative review of state-of-the-art query rewrite methods. Chapter 3 introduces DQRS in detail. Section 3.1 first clarifies the relationship between the formulation of LearnedRewrite proposed in the paper and the released implementation used as the practical reference in this thesis, since this distinction shapes the framing of the remaining method description. Sections 3.2 through 3.4 then describe the state representation, action space, value model design, training procedure, integration with the existing search loop, and implementation details. Chapter 4 reports the experimental evaluation, including the experimental setup, main results on the TPC-H-derived and TPC-DS-derived workloads, ablation studies, an analysis of the observed patterns, and a discussion of the scope and validity of the empirical findings. Chapter 5 concludes the thesis and outlines directions for future work.

Chapter 2

Background

2.1 Database Management System and Query Processing

A database management system (DBMS) is a software system that organizes, stores, and provides controlled access to large collections of structured data. Modern DBMSs handle a wide range of responsibilities, including persistent storage, concurrent access, recovery from failures, and query processing. Among these, query processing is the component most directly relevant to the present thesis, since it is the part of the system that turns user-submitted queries into executable plans and ultimately into results.

A typical DBMS architecture includes a storage manager that controls how data are laid out on disk, a transaction manager that handles concurrent access and recovery, and a query processor that handles the path from input query to returned result. The query processor itself is usually broken into a parser, a query rewriter, an optimizer, and an executor. The work in this thesis sits inside the query rewriter, which transforms the input query before it reaches the optimizer.

The remainder of Section 2.1 zooms in on query processing as a pipeline and on the cost estimates that guide it. Section 2.2 then describes how query rewrite is performed within that pipeline, and the rest of the chapter develops the methodological background needed to study rewrite as a search problem.

2.1.1 Query Processing

When a SQL query is submitted to a DBMS, it goes through a sequence of internal stages before any data are touched. At a high level, these stages are parsing, semantic analysis, query rewrite, query optimization, and query execution. Each stage transforms the query representation incrementally: parsing produces a syntax tree from text, semantic analysis annotates it with type and schema information, query rewrite changes its logical form, optimization picks an execution strategy, and execution produces the result.

Parsing checks that the query is syntactically valid SQL and produces an internal tree representation. Semantic analysis then resolves table and column names against the schema, checks that types are consistent, and rejects queries that reference unknown objects. After this point the query is no longer a piece of text but a structured object that subsequent stages can manipulate.

Query rewrite operates on this structured representation. The rewriter applies one or more transformations that change the form of the query while preserving its semantics. Common targets include subqueries, predicates, joins, and aggregations. Once rewriting finishes, the optimizer takes the rewritten query and chooses concrete execution strategies, such as which join algorithms to use, which access paths to follow, and in which order to perform operations. The executor then carries out these decisions and returns the result to the user.

The rewrite stage is where the present thesis intervenes. Two points are worth emphasizing. First, the optimizer's effectiveness depends on the form of the query it receives, so the choices made during rewrite affect what the optimizer can do downstream. Second, the rewrite stage is also where structured search over alternative formulations becomes possible, since the query is already represented as a tree of relational operations. The rest of this chapter develops the technical background needed to discuss rewrite both as a transformation activity and as a search problem.

2.1.2 Cost Estimation in Query Optimization

Section 1.1.1 introduced cost as an optimizer-side estimate obtained from PostgreSQL EXPLAIN. Because every quality measurement reported in this thesis rests on that quantity, this section describes how the optimizer arrives at it and how the value is retrieved.

An optimizer computes cost without executing the query, by combining two ingredients. The first is a set of cardinality estimates, predictions of how many rows each operator in the plan will produce, derived from statistics the system maintains about the data: table sizes, column value distributions, and the number of distinct values in a column. The second is a cost model that assigns a price to the work each operator performs, expressed in terms of sequential page reads, random page reads and per-row CPU effort. The cost of a plan is the accumulated cost of its operators under the estimated cardinalities.

Two properties follow. First, cost is unitless. PostgreSQL expresses cost in arbitrary units anchored to the price of reading one disk page sequentially, so a cost of 1200 does not correspond to 1200 of anything measurable; costs are meaningful only in comparison with other costs produced by the same optimizer under the same configuration. Second, the estimate can be wrong in a specific way. Cardinality estimation is difficult in the presence of correlated predicates, skewed distributions or stale statistics, and an error in an early cardinality estimate propagates upward through the plan, so the discrepancy is not uniform noise but can be systematic for particular query shapes.

PostgreSQL exposes its cost estimates through the `EXPLAIN` command. Given a query, `EXPLAIN` invokes the planner, produces the plan the system would execute, and returns that

plan together with the estimated cost of each node, without executing the query itself. In this thesis the command is issued in JSON form and the total cost of the root node of the returned plan is taken as the cost of the query. This is the quantity denoted `C_EXPLAIN` and defined formally in Equation 3.1.

Two aspects of how this thesis uses `EXPLAIN` should be made explicit. First, the cost values are obtained externally. Rewriting is performed by Apache Calcite, whose `HepPlanner` applies rules in a specified order and does not itself produce a cost estimate for the rewritten query. Whenever a cost value is needed, the system serializes the current rewrite state back to SQL and issues an `EXPLAIN` call to PostgreSQL over a database connection. Every cost value reported in this thesis therefore comes from a process-external round-trip to the database rather than from an internal computation in the rewriter, and the planner configuration under which those estimates are produced is documented in Section 4.1.1.

Second, `EXPLAIN` should not be confused with `EXPLAIN ANALYZE`. The latter executes the query and reports measured execution time alongside the estimates. `EXPLAIN ANALYZE` is not used anywhere in this thesis, and it could not be used inside the rewrite search: the search generates several candidate states per query, each of which would have to be executed in full, so replacing estimation with execution would increase the cost of rewriting by orders of magnitude and defeat the purpose of the work. All quality measurements reported in Chapter 4 are consequently optimizer-side estimates rather than measured runtime, a limitation stated in Section 4.5.2 and taken up as future work in Section 5.2.

2.2 Query Rewrite Fundamentals

Query rewrite changes the logical form of a query while preserving its semantics. This section describes how rewrite is realized in practice. Section 2.2.1 covers heuristic and rule-based rewrite mechanisms, the dominant approach to rewrite in production systems. Section 2.2.2 introduces the policy tree representation, which makes it possible to talk about alternative rewrite orders as a structured search space rather than as an implicit traversal.

2.2.1 Heuristic and Rule-Based Rewrite Mechanisms

Rule-based rewrite is the most widely used approach to query rewrite in modern DBMSs. The basic idea is straightforward: instead of generating a new query from scratch, the system relies on a fixed set of predefined rules that describe how one query form can be transformed into another. When a rule matches part of the current query, that part is replaced by the rule's target form. The rewritten query is then either passed downstream or used as the input to further rewrites.

Although different systems formalize rules in different ways, their underlying structure is broadly similar. A rule typically consists of a pattern that specifies which fragments of a query or logical plan are eligible for rewriting, a set of validity conditions that must hold for the rewrite to be semantics-preserving, and a transformation that produces the target

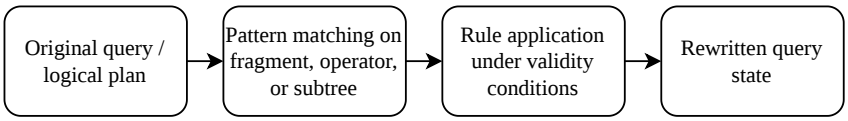

Figure 2.1: Rule-based query rewrite as local matching and replacement.

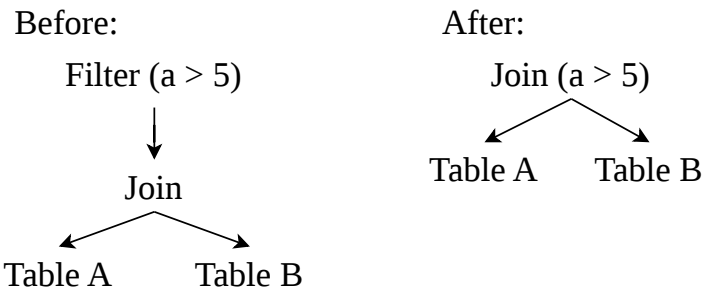

Figure 2.2: Example of merging a filter predicate into a join condition.

form once the conditions are satisfied. Some systems write the matching function explicitly as code, while others express the same idea declaratively through source and destination templates together with attached constraints.

In operation, rule-based rewrite proceeds by pattern matching over the query or its logical representation. The matching target is rarely the whole query at once; more often it is a fragment, an operator, or a subtree. Once a rule matches and its conditions are satisfied, the system applies the corresponding transformation and produces a new query state. Viewed this way, query rewrite is more accurately described as a sequence of local edits than as a single global operation.

Figure 2.1 captures the abstract flow of rule application but does not show what a concrete rule looks like in practice. Figure 2.2 gives one specific example. The rule takes a logical plan in which a filter predicate sits above a join and rewrites it into a plan in which the predicate is incorporated into the join condition. This is one form of filter–join interaction; a closely related rewrite, filter pushdown, instead pushes the predicate below the join onto the input relation it references, and is not the case shown here.

When multiple rules apply at the same point, rule validity alone does not determine what the system should do. The system must also decide the order of rule application. Traditional rule-based rewrite typically manages this order through heuristics. Common strategies include traversing the query in a fixed top-down or bottom-up direction, applying rules in a fixed predefined sequence, or following expert-defined ordering rules tailored to specific transformation families. These strategies are easy to implement and incur little rewrite-time overhead.

The cost of this simplicity is that fixed-order heuristics are brittle in the presence of rule interactions. Applying one rule may enable, disable, or alter the usefulness of subsequent rules, so a sequence that looks good locally may not be the best one available. Pattern

matching can also signal that a rule is applicable without indicating that applying it will be globally useful at this point of the rewrite. As a result, heuristic rewrite is practical and often acceptable, but it does not, by itself, address the larger question of how to navigate the space of possible rewrite sequences.

Several widely used systems are representative of the heuristic and rule-based approach. PostgreSQL [18] is the open-source DBMS used as the executable backend in this thesis, and it includes a rewrite layer that applies fixed rule patterns under predefined orderings. The Volcano optimizer generator [9] is an early framework that organizes rule application as a transformation system over operator trees and influenced many subsequent systems. Apache Calcite [2] provides a query optimization framework with a configurable rule planner; users can add or remove rules and adjust match orders, but the underlying decision procedure remains rule-driven. These systems differ in detail, but they share the same broad approach: rules are defined ahead of time, and rule application is driven by traversal heuristics rather than by query-dependent search.

Once query rewrite is viewed as a sequence of rule applications, it becomes natural to ask what structure those sequences live in. The next subsection introduces the policy tree as one way to organize them.

2.2.2 Policy Trees for Query Rewrite

When rewrite is applied as a sequence of local edits, different choices at each step lead to different successor states. From the same input query, one applicable rewrite produces one rewritten form, while a different choice produces another. Repeating this branching across multiple steps gives rise to a rooted tree structure rather than a single linear sequence. This structure is referred to as a policy tree.

The root of a policy tree corresponds to the original input query. Each non-root node corresponds to a rewritten query state obtained from its parent by exactly one rewrite step. An edge from a parent to a child represents the rewrite step that produced the child. A path from the root to any node, therefore, encodes a concrete rewrite order: the sequence of rewrite steps applied along that path. A leaf node is a query state to which no further applicable rule remains, or to which the search has not yet expanded.

Several properties make this representation useful. Different rewrite orders that share an initial sequence of steps share the corresponding upper-level nodes in the tree, so common prefixes are not duplicated. The space of possible rewrite orders becomes explicit, instead of being implicit in a fixed traversal procedure. Once the rewrite process is organized as a tree of successor states, it becomes possible to reason about which parts of the rewrite space are worth exploring.

Representing rewrite alternatives as a policy tree does not, by itself, determine how the tree should be explored. Once the rewrite space is organized in this form, the next question is how to choose which nodes to expand under a limited search budget. Section 2.3 introduces Monte Carlo Tree Search as one search method designed for exactly this kind of decision space.

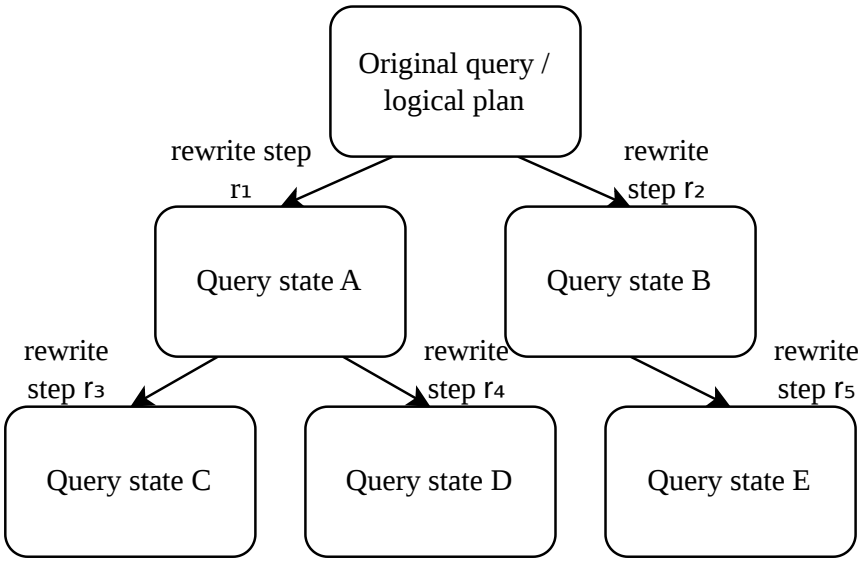

Figure 2.3: A simplified policy tree for query rewrite.

2.3 Monte Carlo Tree Search

MCTS is a tree-search method aimed at decision spaces that are too large to enumerate exhaustively. Instead of constructing the full search tree in advance, MCTS grows the tree incrementally and concentrates effort on branches that look more promising over time. This makes it well suited to combinatorial decision problems with limited search budgets, including game-playing, planning, and structured optimization.

A standard MCTS iteration follows four recurring steps: selection, expansion, simulation, and backpropagation. Starting from the root, the algorithm selects a path through the current tree according to a tree policy, expands the selected node by adding new child nodes, estimates the value of the newly reached state by some form of simulation or rollout, and propagates the resulting signal back along the visited path. The tree and its associated statistics are refined gradually through repetitions of this loop. Figure 2.4 summarizes the four-step iteration.

In the selection step, MCTS begins at the root and repeatedly chooses one child according to the tree policy until it reaches a node that is not yet fully explored or has been chosen for expansion. The purpose of this step is not to revisit the currently best-looking branch, but to balance two competing goals. One is exploitation: revisiting nodes that have produced good outcomes in past iterations. The other is exploration: occasionally trying nodes that have been visited less frequently and whose value is therefore less certain.

Once the selected path reaches a node that can still be expanded, MCTS performs expansion by adding one or more child nodes corresponding to previously unexplored actions or successor states. This is the step that allows the tree to grow incrementally over iterations instead of requiring the entire search space to be generated up front. In some applications, expansion adds only a single child per iteration; in others, multiple children may be added at once.

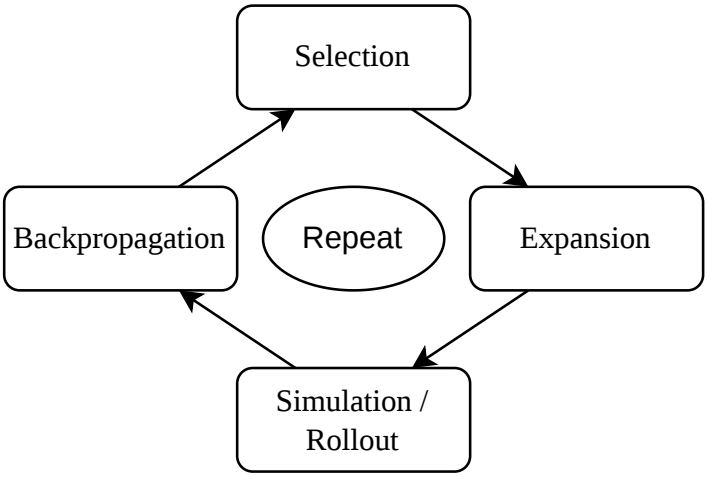

Figure 2.4: A simplified view of the standard MCTS loop.

After expansion, classical MCTS performs a simulation, often called a rollout, from the newly reached node. The rollout produces an approximate estimate of the downstream value of that node without enumerating the rest of the subtree. In the canonical formulation, the rollout policy is intentionally simple, since its role is to give a fast value signal rather than an accurate evaluation. In task-specific variants, the rollout can be replaced by a learned value estimate, a heuristic evaluation function, or even a deterministic computation tied to the application domain.

The signal produced by the rollout is then propagated back along the path that was just traversed, updating the statistics of every visited node. Common statistics include visit counts and accumulated value estimates. This backward update is what lets future iterations reuse the evidence gathered from earlier traversals: a node that has been visited many times with good outcomes will look more attractive to the tree policy in subsequent iterations.

A common way to implement the tree policy is the Upper Confidence Bound applied to Trees (UCT), which adapts bandit-style reasoning to tree search. UCT prefers nodes that currently look valuable while still giving some weight to nodes that have been explored less frequently. A typical form of UCT is

$$\mathrm{UCT}(v_i) = \bar{X}_i + C\sqrt{\frac{\ln N}{n_i}}, \tag{2.1}$$

where $\bar{X}_i$ is the current value estimate of node v_i, n_i is the visit count of v_i, N is the visit count of its parent, and C controls the strength of the exploration term. The first term encourages exploitation by favoring nodes with higher value estimates; the second term encourages exploration by giving more weight to less-visited nodes. In task-specific applications the exact balancing function may be customized instead of following the textbook UCT form.

MCTS is attractive in the context of query rewrite for two reasons. First, the space of possible rewrite orders is combinatorial, and exhaustive enumeration over it is generally too expensive within practical rewrite latency. Second, the policy tree representation introduced in Section 2.2.2 already exposes the kind of structured decision space that MCTS is designed for: a rooted tree where each path corresponds to a sequence of actions and each node carries some value to be estimated. Section 2.5.2 returns to this connection when discussing how learning-guided query rewrite combines MCTS with learned value estimation.

2.4 Deep Reinforcement Learning

The proposed method in this thesis uses a value-based reinforcement learning model to guide rule-class selection during expansion. This section introduces the necessary background. Section 2.4.1 covers the basic framing of reinforcement learning. Section 2.4.2 then introduces Q-learning and the Deep Q-Network (DQN), which together provide the methodological basis for the value model used later in Chapter 3. The treatment is concise and focuses on the elements that recur in the rest of the thesis, rather than on a comprehensive coverage of the field.

2.4.1 Reinforcement Learning Basics

Reinforcement learning (RL) is a framework for sequential decision making, in which an agent interacts with an environment over a series of time steps and learns from the outcomes of its actions. Unlike supervised learning, where each training example is paired with a correct label, RL provides supervision through a reward signal that depends on the agent's choices and the state of the environment. The quality of an action, therefore, depends not only on its immediate effect but also on what happens afterward.

The interaction is described in terms of a small number of recurring elements. At each time step, the agent observes a state s that summarizes the environment's current situation. Based on this state, the agent selects an action a from a set of available actions. The environment then transitions to a new state s' and returns a scalar reward r. The mapping from states to actions used by the agent is called its policy, and the central problem of RL is to find a policy that produces good long-term outcomes.

The standard formal model for this kind of interaction is the Markov decision process (MDP), specified by a state space, an action space, transition probabilities between states, and a reward function. The Markov property requires that the next state and reward depend only on the current state and action, not on the longer history of the interaction. Many practical problems do not satisfy this property exactly, but they can often be modeled as MDPs by enriching the state representation enough to capture the relevant history.

Because rewards arrive at every step, the agent's goal is usually expressed in terms of a cumulative return rather than any single reward. The discounted return at time t is

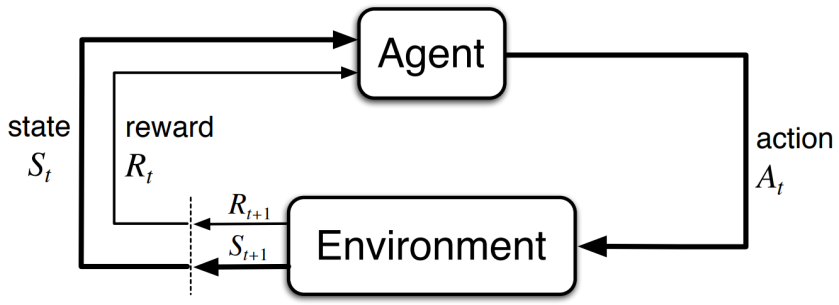

Figure 2.5: Agent–environment interaction in reinforcement learning. Source: Sutton and Barto [23].

defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.2}$$

where $\gamma \in [0, 1)$ is the discount factor. The discount factor controls how much weight is given to future rewards relative to immediate ones. A smaller γ makes the agent more shortsighted; a larger γ makes future rewards almost as important as immediate ones. In practice, γ is treated as a hyperparameter chosen for the task.

Two functions play a central role in value-based RL. The state value function $V^\pi(s)$ is the expected return when the agent starts in state s and follows policy π thereafter. The action-value function $Q^\pi(s, a)$ is the expected return when the agent starts in state s, takes action a, and follows policy π from the next step on. The action-value function is the more useful of the two for the present thesis, since it directly supports comparing alternative actions in the same state.

2.4.2 Q-Learning and Deep Q-Network

Q-learning is a value-based method that learns the action-value function directly from interaction, without requiring a model of the environment. The classical tabular form of Q-learning maintains a table $Q(s, a)$ indexed by every state-action pair and updates each entry incrementally based on observed transitions. After observing a transition (s, a, r, s'), the update rule is

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \tag{2.3}$$

where α is a learning rate. The bracketed quantity is the temporal-difference error, the difference between the current estimate and a target formed from the observed reward together with the best estimated continuation value at the next state. Updates of this form bootstrap from existing estimates rather than waiting for complete returns.

Tabular Q-learning becomes impractical when the state space is high-dimensional or too large to enumerate explicitly. Maintaining a separate value for every state-action pair

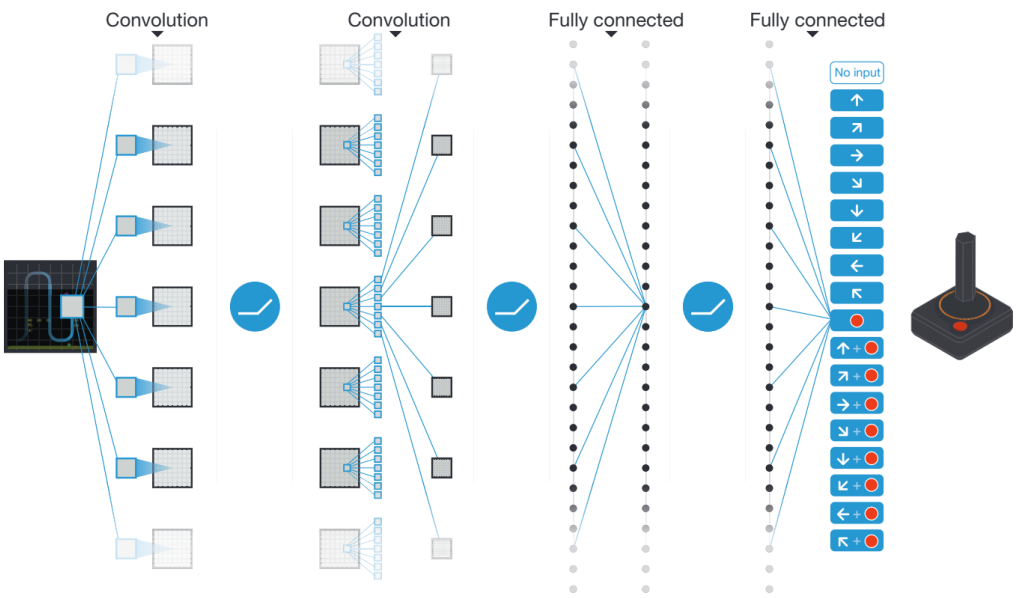

Figure 2.6: The Deep Q-Network architecture proposed by Mnih et al. [16].

is no longer feasible, and similar states that have not been visited cannot benefit from one another's estimates. In settings of this kind, the tabular representation is replaced by a function approximator that maps state-action pairs (or states alone, with one output per action) to estimated values.

The Deep Q-Network (DQN) replaces the Q-table with a neural network that takes a state representation as input and outputs estimated action values. The network is trained to minimize the difference between its current predictions and target values formed in the same way as in tabular Q-learning. In a common form, the target for a transition (s, a, r, s') is

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-), \tag{2.4}$$

where θ^- denotes the parameters of a separate target network. The training loss is typically the mean squared error between the network's prediction $Q(s, a; \theta)$ and the target y. The use of a function approximator allows the model to generalize across similar states and to handle action spaces that would be impossible to enumerate as a table. Figure 2.6 shows the architecture proposed in the original DQN paper, where the network maps a state to one Q-value per action.

Two design choices are central to DQN training stability. The first is experience replay. Instead of training only on consecutive interactions, the agent stores past transitions in a replay buffer and samples mini-batches from it during training. This breaks the strong temporal correlation between consecutive transitions and improves data reuse, since the same transition can contribute to learning many times after it is collected.

The second is the use of a separate target network for computing the bootstrap target in Equation (2.4). The target network's parameters θ^- are updated only intermittently, either by periodic copies from the online network or by slow moving averages. Without

this separation, the network would be trained against a target that shifts at every gradient step, which can make optimization unstable. Together with experience replay, the target network is one of the standard mechanisms that make neural Q-learning practical.

DQN is particularly suitable when the action space is finite and discrete. Because the network outputs one value per available action at a given state, the agent can compare all actions in a single forward pass and choose among them by greedy selection or by ranking. This action-comparison view is the core reason DQN is used in this thesis: the rule-class selection problem at expansion time is naturally a discrete action-ranking problem at each search state. The concrete state representation, action design, reward construction, and training objective used in this thesis are introduced later in Chapter 3.

2.5 Modern Query Rewrite Techniques

The previous sections covered the technical machinery used in the thesis: rule-based rewrite, the policy tree representation, MCTS, and DQN. This section turns to existing approaches to query rewrite that build on or extend this machinery. The goal is to position the present thesis within the broader literature, not to give an exhaustive review. Section 2.5.1 covers automated rewrite approaches that go beyond manually written rules. Section 2.5.2 covers learning-guided rewrite, where the rule set remains fixed and what is learned is the decision policy over rule selection and ordering. Section 2.5.3 covers the more recent line of LLM-driven rewrite, including both rule-constrained LLM-assisted systems and generative LLM-driven systems. Section 2.5.4 closes with a comparative summary.

2.5.1 Automated Query Rewrite

Traditional heuristic rewrite depends heavily on rules that are designed and ordered by hand, which limits how broadly the approach scales. To reduce this manual effort, several lines of work have introduced more automation into the rewrite process. The shared theme is that rules, rule discovery, or rewrite generation can be supported by tools rather than left entirely to expert judgment.

One direction focuses on automated rule discovery. WeTune [26] is representative of this direction. It searches for previously unknown query rewrite rules and verifies their correctness automatically, expanding the rule set available to a rewrite system without requiring an expert to author each rule by hand. The contribution of this line of work is largely orthogonal to the rule application stage: it grows the rule pool that subsequent rewrite systems can draw from.

A second direction is human-centered rewrite support. QueryBooster [1] takes this view. It is a middleware service through which end users can describe and apply rewrite rules without modifying the underlying DBMS. The system focuses on making rewrite accessible rather than on automating decisions, and contributes infrastructure for rewrite rule authoring and deployment.

A more aggressive direction moves beyond predefined rules entirely. SlabCity [5] uses program synthesis to generate semantically equivalent SQL queries for whole-query optimization, rather than applying local transformations from a fixed rule set. This shifts the problem from "which rule to apply where" to "which equivalent query to produce", at the cost of a substantially larger search problem.

Compared with traditional heuristic rewrite, these automated approaches reduce dependence on manually engineered rules and broaden the space of available transformations. They do not, however, directly address the question of how to choose among alternative rewrite sequences for a given query. Whether more rules are discovered, exposed through middleware, or replaced by synthesized queries, the system still needs a way to decide what to do at each rewrite step. This is where the next line of work, learning-guided rewrite, becomes relevant.

2.5.2 Learning-Guided Query Rewrite

A different line of work shifts the focus from rule availability to rule selection. The premise is that, given a set of rules, the more important question is which rules to apply and in what order for a particular query. Approaches in this line use learning to guide this decision instead of relying on fixed heuristics, on the grounds that decisions tied to a specific query and its current state are difficult to capture by hand-tuned rules. The rule set itself remains fixed and explicit; what is learned is the decision policy over how to use it.

LearnedRewrite [30] is an early representative of this direction. It models query rewrite as a sequential decision-making problem over a fixed rule set. Alternative rewrite orders are organized as a policy tree, exploration over the tree is performed by Monte Carlo Tree Search, and a learned cost estimator provides a value signal that guides the search toward more promising rewrite states. Combining MCTS with learned estimation lets the system focus search effort on promising parts of the rewrite space while keeping the overall procedure rule-constrained.

LearnedRewrite is also the framework on which the present thesis is built. A more detailed description, together with the relationship between the formulation proposed in the paper and the released implementation that serves as the practical reference in this work, is given in Section 3.1. The current section keeps the description at survey level and treats LearnedRewrite as one entry in the broader landscape of learning-guided query rewrite.

Beyond query rewrite specifically, reinforcement learning has been applied to a range of database optimization problems. Examples include join order selection [15, 28], end-to-end learned optimizers [14], and learned hint selection that steers an existing optimizer [13]. While these works do not target query rewrite directly, they share the methodological premise that complex optimization decisions in DBMSs can be framed as state-dependent action selection problems. The same premise underlies the rule-class selection formulation developed in this thesis.

2.5.3 LLM-Driven Query Rewrite

The recent emergence of large language models (LLMs) has opened a new line of work in query rewrite that differs in character from the rule-constrained learning-guided systems described above. The central change is that one or more decision points in the rewrite process are handled by an LLM rather than by hand-tuned heuristics or by a value model trained on rewrite transitions. Within this broad direction, two subclasses can be distinguished by how strongly they retain explicit rule structure.

The first subclass uses LLMs as a decision component while keeping the rule set fixed and explicit. Rewrite generation itself is still constrained by the underlying rule-based system; what the LLM contributes is rule selection and ordering. LLM-R^2 [11] takes this approach by using a large language model to recommend appropriate rewrite rules for a given query, supported by contrastive query representations and in-context demonstrations. R-Bot [22] develops the idea further by combining evidence retrieval with step-by-step language-model reasoning over rule selection and arrangement, so the rule selection decision is informed by retrieved supporting information rather than by the language model alone. In both systems, the rules themselves are not generated; the language model decides which rules from the existing set to use and in what order.

The second subclass takes a more aggressive approach: rewritten SQL is produced by the language model itself, without the constraint of a predefined rule set. The rewrite is generated rather than selected from existing transformations. This raises questions of correctness, executability, and equivalence that do not arise in rule-constrained systems, since there is no guarantee that the generated query is semantically equivalent to the input. GenRewrite [12] is one of the earlier representatives of this subclass. It augments LLM-based rewrite generation with natural-language rewrite rules that act both as hints to the model and as a way to transfer knowledge between queries. To address correctness issues that arise from purely generative rewriting, GenRewrite uses a counterexample-guided correction loop that iteratively repairs semantic and syntactic errors. The pipeline is therefore not simply LLM rewriting in isolation; generation, hinting, correction, and verification are combined into a controlled workflow.

Other systems extend this generative direction toward more system-level integration. LITHE [4] is a query-space rewriting assistant that combines prompt-based generation with database-sensitive guidance, semantic verification, and DBA-facing advisory output. QUITE [21] pushes the trend further with a training-free, feedback-aware multi-agent framework that refines rewrites using database feedback and external tools. These systems treat rewrite as a multi-step interaction with the database rather than as a single generation step, and they introduce additional safeguards to compensate for the lack of explicit rule structure.

E^3-Rewrite [27] takes the integration further by tying rewrite generation to execution-aware end-to-end optimization. It defines a useful rewrite by three properties at once: executability, equivalence, and efficiency. Execution-plan-aware context construction and reinforcement learning objectives aligned with these three properties are combined to train a model that targets practically meaningful rewrites rather than abstract equivalent queries.

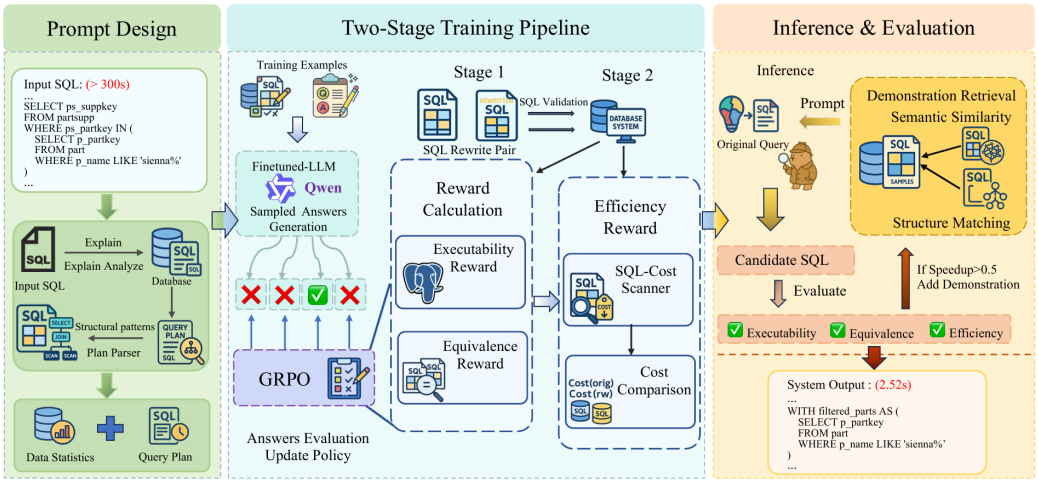

Figure 2.7: System overview of E^3-Rewrite [27].

The shift here is from "produce an equivalent query" toward "produce a query that is also executable and efficient under the target system". Figure 2.7 illustrates the system, showing how execution-plan-aware context construction interacts with the three-property training objective.

Compared with the learning-guided line described in Section 2.5.2, LLM-driven rewrite operates with much greater flexibility but also faces a different set of concerns. Generation quality, semantic validity, and execution awareness become central questions, since neither subclass is constrained to a fixed rule set with verified equivalence properties in the strict sense. The two lines are complementary rather than competing: one targets low-overhead rule application within a fixed rule set, while the other extends rewriting beyond fixed rules at the cost of additional safeguards. Whether language-model-based preference could be applied at the finer granularity studied in this thesis, namely a ranking decision at every expanded node of a policy-tree search rather than a single decision per query, is a distinct question from the query-level selection these systems perform; it is taken up as a future work direction in Section 5.2.

2.5.4 A Comparative Analysis of the State-of-the-art

The three lines surveyed in this section reflect different starting points for query rewrite research. Heuristic rule-based rewrite is inexpensive to run and broadly applicable but does not address query-dependent rule ordering. Automated approaches reduce manual rule engineering but leave the rule application decision largely untouched. Learning-guided systems treat rule ordering itself as an optimization target, with LearnedRewrite as an early representative. LLM-driven systems take a much broader view of what rewrite can be, at the cost of additional concerns around correctness and execution awareness.

The present thesis is positioned within the learning-guided line. It does not propose a new framework for query rewrite, nor does it attempt to compete with LLM-driven

approaches on flexibility. Instead, it studies a focused efficiency question inside the rule-constrained learning-guided setting represented by LearnedRewrite: how should rule-class selection during expansion be guided so that rewrite-search overhead is reduced while degradation in the released implementation's estimator-side cost-reduction signal is limited? This question is formalized operationally in Section 1.3 and taken up methodologically in Chapter 3.

The background needed to make the rest of this thesis self-contained has now been covered. The next chapter develops the proposed method, DQRS, within the search-based rewrite framework introduced here.

Table 2.1: Comparison of representative approaches related to query rewrite. Entries in the Strengths and Limitations columns describe properties of the *approach*, not of the queries it produces; *overhead* in this table refers to the cost of running the rewrite procedure itself. The two senses in which *efficient* is used in this thesis are distinguished in Section 1.1.1.

Category	**Representative Systems**	**Strengths**	**Limitations**
Traditional Heuristic Query Rewrite	PostgreSQL; Volcano; Apache Calcite	Conceptually simple, low rewrite-time overhead, and easy to deploy in straightforward cases	Fixed rule orders limit search coverage and the handling of inter-rule dependencies
Automated Query Rewrite Beyond Manual Heuristics	WeTune; QueryBooster; SlabCity	Reduces manual effort and broadens rewrite support	Does not fully solve query-dependent search and decision-making
Learning-guided Query Rewrite	LearnedRewrite	Introduces query-dependent search and learned guidance for rewrite ordering	Efficient action selection in large policy spaces remains challenging
Reinforcement Learning in Database Optimization	ReJOIN; RTOS; Neo; Bao	Provides methodological support for RL-style state-dependent optimization decisions	Mostly targets join ordering or optimizer control rather than query rewrite directly
LLM-Driven Query Rewrite	GenRewrite; LLM-R^2; R-Bot; LITHE; QUITE; E^3-Rewrite	Greatly expands flexibility, semantic coverage, and execution-aware rewriting	Generation quality, semantic validity, executability, and equivalence remain central concerns

Chapter 3

Introduction of DQRS

The previous chapter introduced the technical machinery underlying this thesis: rule-based rewrite, the policy tree representation, MCTS, and DQN. This chapter develops DQRS, the method proposed in this thesis to guide expansion-time rule-class selection within search-based query rewrite. The chapter is organized as follows. Section 3.1 clarifies the relationship between the formulation of LearnedRewrite proposed in the paper and the released implementation that serves as the practical reference here, since this distinction shapes the rest of the chapter. Section 3.2 describes the state representation and the rule-class-level action space. Section 3.3 presents the value model, its training objective, and how the trained model is integrated into the search loop at expansion time. Section 3.4 describes the implementation, including how training data are collected, how the model is trained offline, and how it is deployed for online inference.

3.1 LearnedRewrite Framework: Formulation in the Paper and Released Implementation

This section serves a different purpose from a typical background section. It does not survey LearnedRewrite as one entry in the literature; that survey-level treatment is in Section 2.5.2. Instead, this section describes LearnedRewrite as the concrete framework on top of which DQRS is built. Section 3.1.1 describes the system as proposed in the original paper. Section 3.1.2 describes the released single-rule implementation, which is the actual baseline used in this thesis. Section 3.1.3 states the implications of the gap between the two for the framing of this work.

3.1.1 Formulation in the Paper

LearnedRewrite [30] formulates rule-based query rewrite as a search problem, not as a fixed-order procedure. Given an input SQL query and a fixed set of rewrite rules, the system represents alternative rewrite orders as a policy tree (Section 2.2.2), explores promising

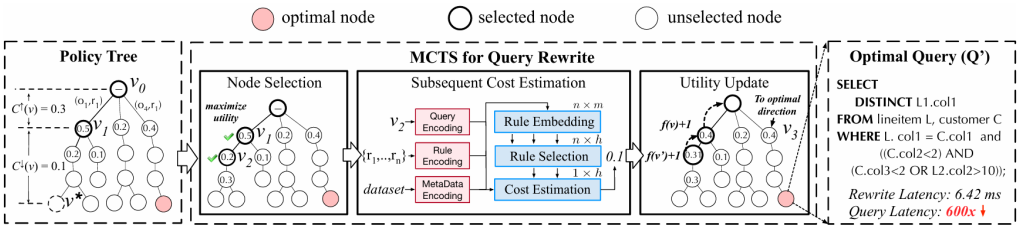

Figure 3.1: Overview of the LearnedRewrite framework proposed in the paper [30].

rewrite states using MCTS (Section 2.3), and uses a learned model to estimate the value of a state more reliably than what a single immediate cost reduction would suggest. The output is a rewritten query selected from the explored portion of the tree.

The policy tree is rooted at the original query. Each non-root node corresponds to a rewritten state obtained from its parent through the application of one rewrite rule. Each path from the root to a node thus represents one concrete rewrite order, and the leaves of the tree are query states from which no further rewrite is attempted. The space of paths grows quickly with the number of applicable rules, which is why exhaustive enumeration is impractical and why search is needed.

MCTS provides the mechanism that allocates limited search effort over this tree. At each iteration, the algorithm selects a node according to the tree policy, expands it by applying available rewrite rules to generate candidate children, evaluates the new states, and updates node statistics along the path. Over many iterations, the search concentrates on parts of the tree that look more promising while still allowing some exploration of less-visited branches. Within a fixed rewrite budget, this lets the system focus on the rewrite orders most likely to produce useful rewritten queries.

The learned cost estimator is the third component and is one of the central contributions of the paper. The motivation is that the value of a rewrite node is not fully captured by its immediate cost reduction. A rewrite with limited local gain may lead to descendants with much larger benefits, while a rewrite that looks locally favorable may lead nowhere. To capture this, the paper introduces a learned model that estimates the downstream value of a rewrite state, separating it conceptually into a previous-cost-reduction component and a subsequent-cost-reduction component. The learned estimator provides a value signal that guides MCTS more accurately than a purely local cost-based signal would. Figure 3.1 shows the overall framework, with the three components and their interaction in the search loop.

3.1.2 Released Implementation

The system used as the practical reference in this thesis is the publicly released single-rule implementation of LearnedRewrite [29]. The released code differs from the system proposed in the paper in one important way: it implements the policy-tree search structure described in Section 3.1.1, but it does not implement the learned cost estimator. Cost evaluation in the released code is provided by PostgreSQL EXPLAIN. The released implementation

therefore corresponds to a structurally faithful but learning-guidance-stripped instantiation of the framework proposed in the paper.

A second feature of the released code that is relevant to this thesis is how candidate generation during expansion is organized. The release does not expose a fully fine-grained per-rule action space; instead, it organizes candidate rewrite generation through rule classes. Each rule class corresponds to a category of rewrite rules that target the same family of operators or transformation patterns. Specifically, the released single-rule implementation uses seven rule classes: `rule_filter`, `rule_join`, `rule_agg`, `rule_project`, `rule_cal`, `rule_orderby`, and `rule_union`. Expansion at a selected node proceeds by checking which of these rule classes are applicable to the current relational state and then attempting rewrite through each applicable class.

Each candidate rule class, when triggered, runs through a fixed sequence of operations inside the release. The current relational state is passed to the corresponding rule routine, which applies the transformation if its conditions are satisfied and produces a candidate child query. The candidate must then pass an executability check via PostgreSQL EXPLAIN. If it passes, the cost of the candidate is obtained from EXPLAIN, and the candidate is accepted as a child node only if its cost is positive and does not exceed the cost of the original query. Accepted children become new nodes in the policy tree.

Because the learned cost estimator is absent, the only value signal available in the released search is the EXPLAIN cost itself. Node value during search is therefore tied directly to the optimizer's static cost estimate of the corresponding rewritten query. This is a meaningful difference from the system proposed in the paper, where the learned estimator was specifically designed to provide a signal richer than per-state immediate cost. In the released code, the search procedure operates without access to that downstream-value signal.

The released code does include some lightweight gating during expansion. Rule applicability checks reject rule classes whose required operator pattern is not present in the current state, and the cost-based filtering described above rejects candidates whose cost is non-positive or exceeds the original query cost. These mechanisms are coarse and query-agnostic in the sense discussed in Section 1.3: they operate on per-candidate properties rather than on the relationship between the current state and the candidate's downstream potential.

3.1.3 Implications for This Thesis

The implications of this gap between the paper and the released code for this thesis are direct. The baseline used throughout this work is the released single-rule implementation, not the system proposed in the paper in its full form. Comparisons of DQRS against the baseline used in this thesis therefore mean comparisons against the released implementation, which lacks the paper's learned cost estimator. This choice is deliberate: the released code is the publicly available reference, and DQRS is implemented as a module-level modification on top of that codebase, so comparing DQRS against the released code under identical conditions isolates the effect of the proposed modification.

A second implication is that DQRS does not replace the paper's learned cost estimator. The two operate at different points in the search loop. The paper's estimator operates at the selection stage, providing a value estimate for a candidate node so that MCTS can decide which node to traverse next. DQRS operates at the expansion stage, ranking the rule classes applicable to the current state so that only a subset is forwarded for candidate generation. The two interventions target different decision points and are not in direct competition by design. Whether they would compose constructively in a single system, interfere with each other, or simply duplicate effort is an empirical question that requires reproducing the paper's estimator, and is outside the scope of this thesis. The present work evaluates only the expansion-side intervention, since that is the side compatible with the released code.

A third implication is on the scope of the empirical claims in this thesis. Reproducing the paper's learned cost estimator is outside the scope of this work. Whether the same magnitude of efficiency gain reported in Chapter 4 would carry over to a full reproduction of the system proposed in the paper remains an open question. This scope condition is revisited explicitly in the scope and validity considerations in Section 4.5.

3.2 State Representation and Action Space

Before describing the value model itself, we fix three related but distinct concepts that the rest of this chapter relies on. A *state* is the rewrite situation the search has reached at a given moment, represented internally by a relational tree at a node of the policy tree, the same data structure the rewrite framework already operates on. A *feature vector* is a fixed-length numeric encoding of a state, produced by a hand-crafted encoder; the Q-network consumes this vector rather than the relational tree directly. An *action* is the selection of one rewrite rule class, drawn from the seven rule classes exposed by the released implementation; the seven actions form the action space over which the Q-network outputs values.

Section 3.2.1 describes the state, the encoder, and the resulting 32-dimensional feature vector. Section 3.2.2 describes the action space and the legal-action mask that suppresses inapplicable rule classes at the current state.

3.2.1 State Representation

A learned value model imposes a requirement that the search procedure's own data structures do not satisfy. A neural network takes input of a fixed dimension, expressed as real numbers. A rewrite state is neither: it is a tree whose shape, depth, size and operator composition vary from one query to the next and from one rewrite step to the next within the same query. Encoding is therefore not an optional preprocessing convenience but a precondition for using a learned model at this point in the search at all. The role of the encoder is to map a variable-shaped relational tree onto a fixed-length numeric vector,

retaining enough information for the decision the model is actually asked to make, which here is a ranking over at most seven rule classes rather than a reconstruction of the query.

DQRS treats rule-class selection as a state-dependent decision problem, so the feature vector that the value model consumes must distinguish meaningfully between different rewrite states while remaining cheap enough to compute repeatedly during online search. The encoder is not applied to the raw SQL text. Each search state is represented internally by the current relational tree, and the encoder reads from this relational tree directly. The relational tree is the same internal representation that the rewrite framework already operates on: rule applicability checks and rewrite execution both also work over the relational tree, so the encoder shares the same structural view of the state as the rest of the decision process. No additional parsing step is needed between the search loop and the learned model.

The representation is intentionally compact. It is a fixed-length feature vector summarizing structural and coarse cost-related properties of the current state, rather than a learned embedding over the relational tree. A learned embedding could, in principle, capture richer signals, but it would also add a separate representation-learning subproblem and increase per-step inference cost. The hand-crafted compact representation used here is a more conservative choice that fits the scope of the present thesis.

We now specify the encoder. The current rewrite state, represented internally as a relational tree, is encoded as a fixed-length 32-dimensional feature vector. Of these 32 dimensions, fourteen carry signal: eight structural-shape features (tree height, total relational-node count, and counts of joins, filters, aggregates, scans, sorts, and projects), three coarse size-and-cost features (output field count, a log-scaled row-count estimate, and a log-scaled cost estimate from the optimizer), and three operator-presence indicators (join, aggregate, sort). The structural-shape features describe the overall composition of the relational tree, the size-and-cost features give the value model a sense of state scale when comparing rule classes, and the presence indicators flag whether key operator categories appear in the tree at all. The input layer of the Q-network is fixed at 32 dimensions to maintain a stable Open Neural Network Exchange (ONNX) input signature across feature-design iterations during development; the additional 18 dimensions are held at zero across all states, so they carry no information and contribute nothing to the pre-activation of the first hidden layer.

To illustrate the feature encoding, consider a simple filter-only rewrite state such as

```
SELECT * FROM part WHERE p_partkey > 815986 AND p_partkey > 958249;
```

In the Calcite relational representation used by the implementation, this state has the shape `Project–Filter–TableScan`. Ignoring the 18 padded zero dimensions, the active part of the encoded vector is

$$[3,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 9,\ 0.693,\ 8.761,\ 3,\ 0,\ 0,\ 0],$$

where the entries correspond, in implementation order, to tree height, join count, filter count, aggregate count, scan count, sort count, project count, output-field count, log-scaled row-count estimate, log-scaled optimizer cost, total relational-node count, and the

three presence indicators for join, aggregate, and sort. In one recorded transition from the training data, applying a filter-related rule keeps the same relational-tree shape but changes the optimizer-cost feature, giving a next-state vector whose active part is

$$[3,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 9,\ 0.693,\ 8.728,\ 3,\ 0,\ 0,\ 0\,].$$

The two vectors differ in exactly one position. This example makes concrete what the encoder must be able to represent: a purely structural encoding, built only from operator counts and tree shape, would assign these two states identical vectors and would therefore be unable to tell them apart, even though the second is the outcome of a rewrite that the optimizer estimates to be cheaper. Including cost-derived features is what allows the encoder to separate states that a rewrite has changed in estimated quality but not in shape. The converse also occurs: rewrites that introduce or remove operators change the corresponding structural counts and presence indicators, and for a subquery-to-join rewrite of the kind illustrated in Section 1.2 the encoder would reflect the structural change through features such as join count, total relational-node count, tree height, and the join-presence indicator, while the EXPLAIN-derived cost feature would reflect the optimizer's estimate for the rewritten state.

The representation is intentionally modest in scope. It does not attempt to reconstruct the full semantics of the query or to encode every detail of the relational tree. Two considerations justify this choice. First, the value model is not asked to choose among arbitrary rewrites; it only ranks legal rule classes at the current state, which is a low-dimensional decision over a fixed set of seven options. A lightweight summary of state structure is sufficient information for that decision. Second, the model is invoked repeatedly during online search, so the encoding must remain inexpensive. A fixed-length hand-crafted feature vector keeps the in-process part of this cost small: thirteen of the fourteen active features are obtained from a single traversal of the relational tree, and the forward pass through a network of this size is inexpensive. The encoding is not free overall, however. The remaining feature is the optimizer's cost estimate for the current state, which the implementation obtains through one PostgreSQL EXPLAIN call per expanded node, a call the released baseline does not make. This call is issued once per expansion rather than once per candidate, so its cost is amortized over the k candidates that follow, but at small k it is of the same order as evaluating one candidate. It is included in the rewrite latency reported in Chapter 4.

Once the state is encoded, the next question is what the model is asked to choose. Section 3.2.2 defines the action space and the legal-action masking that constrains the model's output to rule classes applicable at the current state.

3.2.2 Action Space and Masking

An action in DQRS is the selection of one rewrite rule class. The action space is therefore not the space of final rewritten SQL queries, complete rewrite paths, or fine-grained operator-level rewrite instances; it consists of seven rule classes, the same seven exposed by the released implementation: `rule_filter`, `rule_join`, `rule_agg`, `rule_project`,

`rule_cal`, `rule_orderby`, and `rule_union`. The value model is trained to estimate action values over this fixed seven-dimensional action space. Although the action space has only seven entries, each entry is a coarse rule class rather than a single concrete rewrite rule. Table 3.1 reports the number of concrete Calcite rule instances grouped under each action in the implementation, as registered in the rewriter's internal rule-class map. In total, the seven rule-class actions cover 57 concrete rule instances registered across the seven classes. A DQRS action therefore selects a rule group; the existing HepPlanner is then configured with the concrete rules in that group and attempts to rewrite the current relational state. This design keeps the learned action space small while preserving access to the underlying rule inventory exposed by the released implementation.

Table 3.1: Rule-class actions and the number of concrete Calcite rule instances registered under each class in the released implementation.

Action	Operator class	Concrete rules
`rule_filter`	`Filter`	13
`rule_join`	`Join`	10
`rule_agg`	`Aggregate`	9
`rule_project`	`Project`	9
`rule_cal`	`Calc`	2
`rule_orderby`	`Sort`	7
`rule_union`	`Union`	7
Total	—	57

This grouped level of granularity is chosen to match the way expansion is organized in the released code. Candidate rewrites are generated by activating one rule class and allowing the existing rewrite routine to try the concrete rules inside that class, rather than by exposing each concrete rule instance as a separate action. A rule-class-level action space therefore aligns the learning problem directly with the actual branching structure of the rewrite framework, and the value model can be inserted at the existing expansion hook without modifying candidate generation itself. Two alternative granularities were considered but not adopted in the present work. A finer per-instance action space would expose each of the 57 concrete Calcite rule instances reported in Table 3.1 as a distinct action, giving the value model direct ranking control over which low-level rewrite to attempt. Three considerations argued against this choice in the current setting. First, the released candidate-generation interface does not expose per-instance hooks at expansion time; concrete rules are activated together via the rule-class map, so a per-instance action space would require modifying the candidate-generation routine itself rather than only the expansion-time selection step that DQRS targets. Second, a 57-way action space enlarges the Q-network output and increases the per-state applicability-checking cost; under the offline transition dataset described in Section 4.1.6, supervision per action becomes substantially sparser, which weakens Q-value estimates for less-frequent concrete rules. Third, many concrete rules within the same class produce overlapping or near-redundant rewrites at the candidate-generation level, so the additional granularity may not translate into

proportionally stronger ranking signal.

A coarser binary action space, in which a single Q-value decides whether to expand a node at all, was also considered. This action space cannot distinguish between rule classes and therefore cannot guide which rule class to forward to the rewrite routine. It would reduce DQRS to a learned node-level gate on top of the existing per-candidate filtering in the released implementation, which the synthesis in Section 1.3 already identifies as coarse and query-agnostic. A binary node-level gate does not address the limitation that motivates DQRS in the first place. The selection of the seven-action rule-class granularity is therefore an alignment choice driven by the structure of the released candidate-generation interface and by the scope of the offline transition data, not a choice empirically validated against the two alternatives above. The alternatives are described here to make the design space explicit. An empirical comparison against a per-instance action space would require modifying the released candidate-generation interface and re-collecting transitions at the per-instance level; this comparison is identified as future work in Section 5.2, and the corresponding scope condition is recorded as a scope and validity consideration in Section 4.5.4. The seven rule classes are not all valid choices at every state. A rule class is applicable only when the current relational tree contains an operator pattern compatible with that class. To enforce this, DQRS computes a legal-action mask at each state by checking the current relational tree against the operator class associated with each rule class. The check is recursive over the relational tree: if the required operator pattern is found anywhere in the tree, the corresponding entry in the mask is marked legal; otherwise, it is marked illegal. The result is a Boolean mask over the seven rule classes.

During inference, the value model still produces Q-value estimates over the full seven-dimensional action space, but ranking is performed only after the mask is applied. Illegal actions are suppressed by assigning them a large negative value before ranking, so they are excluded from the top-k selection regardless of their predicted Q-value. When the number of legal rule classes is smaller than the configured top-k, the effective k is reduced to the number of legal actions. Without this masking step, the model could rank rule classes that are not applicable to the current state, producing a top-k subset that is inconsistent with what the rewrite engine can actually execute. The mask therefore defines the effective decision space over which the learned ranker operates and clarifies the model's role: it ranks among existing options rather than proposing new ones.

3.3 DQN-Guided Rule-Class Selection

Before describing the value model itself, the role it plays in DQRS should be made precise. The model is implemented in the standard DQN form, with a Q-network trained against TD targets and a separate target network. Training is performed offline on a fixed dataset of rewrite transitions exported from the released search procedure rather than on an online replay buffer fed by ongoing interaction; the dataset plays the role of a static replay source, and no behavior-policy data are collected after training begins. The standard DQN background, including experience replay as a stability mechanism, is summarized in Section 2.4.2. For this reason, it is referred to throughout this thesis as "DQN-guided".

Within the released implementation, however, the role of this model is narrower than what a fully instantiated DQN-based planner would play. The released code provides only per-step EXPLAIN-cost signals and does not expose a downstream-value signal of the kind described in the LearnedRewrite paper. The Q-network in DQRS is therefore trained with the standard DQN target but on a transition-level reward that reflects immediate cost change. Bootstrap targets propagate value information across recorded transitions whose next state has not been empirically confirmed to be terminal, providing a learned ordering signal beyond the per-transition reward. The strength of this propagation in practice depends on dataset coverage rather than on the discount factor: the model can only sharpen its value estimate at states that appear in the recorded transitions, which under the released search procedure are concentrated near the actual rewrite trajectories the original system explores. The model's operational role at inference time is action ranking over the seven rule classes at a single state, which is consistent with the use of a discrete action-value function. The implications of this design choice for what the model can and cannot do are revisited in Sections 3.3.4 and 4.5.

3.3.1 Reward Signal and Transition Construction

DQRS uses a transition-level reward, not a reward tied to the final rewritten query. Let s_t denote the current rewrite state at a policy-tree node, let a_t denote the selected rule-class action, and let s_{t+1} denote the accepted child state produced by applying that action. Let $q(s)$ be the executable SQL string associated with rewrite state s, and let

$$C_{\text{EXPLAIN}}(s) \;=\; \text{TotalCost}(\text{EXPLAIN}_{\text{JSON}}(q(s))) \tag{3.1}$$

denote the PostgreSQL EXPLAIN total-cost estimate for that state. The notion of cost and the mechanism by which this value is obtained are described in Section 2.1.2. The same estimator is used by the released implementation itself when deciding whether to accept a generated child, so the reward signal introduces no quality criterion that the baseline does not already apply. The raw transition reward recorded for DQN training is defined as

$$r_t^{\text{raw}} \;=\; C_{\text{EXPLAIN}}(s_t) \;-\; C_{\text{EXPLAIN}}(s_{t+1}) \tag{3.2}$$

Thus $r_t^{\text{raw}} > 0$ means that the selected rule class produced a local cost reduction from the parent state to the child state, while $r_t^{\text{raw}} \leq 0$ means that the accepted child did not improve the EXPLAIN cost relative to its immediate parent. Because the same PostgreSQL EXPLAIN cost is also the value signal used by the released implementation, the reward is aligned with the surrounding system's estimator-side quality signal.

For a concrete example of $C_{\text{EXPLAIN}}(\cdot)$, consider the filter-only state used in Section 3.2.1:

```
SELECT * FROM part WHERE p_partkey > 815986 AND p_partkey > 958249;
```

For this query state and for the accepted child state produced by the first rule-class application recorded in the training data, the PostgreSQL EXPLAIN total-cost estimates are

$C_{\text{EXPLAIN}}(s_t) = 6380.00$ and $C_{\text{EXPLAIN}}(s_{t+1}) = 6171.67$. Applying Equation 3.2 to these two estimates yields a raw transition reward of

$$r_t^{\text{raw}} \;=\; 6380.00 - 6171.67 \;=\; 208.33$$

for this transition, recording that the selected rule class produced a positive local EXPLAIN-cost reduction at this step. Both EXPLAIN estimates come from the same EXPLAIN call that the released rewrite system uses internally to evaluate candidate children, which keeps the reward signal aligned with the surrounding system's estimator-side quality signal.

The child-acceptance condition in the released implementation is different from the transition-reward definition: a child is accepted when

$$0 < C_{\text{EXPLAIN}}(s_{t+1}) \leq C_{\text{EXPLAIN}}(s_0), \tag{3.3}$$

where s_0 is the original query state. Therefore, an accepted transition can still have $r_t^{\text{raw}} < 0$ if the child is more expensive than its immediate parent but remains no more expensive than the original query. This is consistent with the recorded transition CSV used for DQN training, which contains both positive and negative transition rewards.

This transition-level reward should be distinguished from the query-level cost reduction reported in the experimental evaluation in Chapter 4. The query-level cost reduction is the difference between the cost of the original query and the cost of the final rewritten query selected by the search; it is a single number per query and characterizes overall rewrite quality. The reward used during training, by contrast, is a local per-step signal between a parent and its generated child. The two quantities are related but not identical, and they serve different purposes: the per-step reward provides supervision for training the value model, while the query-level cost reduction is the end-to-end metric for evaluation.

Each training sample is represented as a transition tuple. At a minimum, a transition contains the encoded current state, the selected rule class as the action, the observed reward, the encoded next state, and a terminal flag. The state and next state are stored in the same 32-dimensional feature representation described in Section 3.2.1. The terminal flag requires a separate design decision because the rewrite search is budgeted. Two interpretations of "terminal" are possible. The first is to mark a transition as terminal when the search budget is exhausted before the child state is expanded further. This interpretation was considered and rejected: budget exhaustion is an external truncation of the search process, not evidence that the child state has no valid rewrite successors. Treating budget-truncated states as terminal would suppress the bootstrap term in the DQN target for states whose continuation value is merely unobserved, mixing search-budget control with rewrite-state terminality.

The adopted interpretation is therefore stricter. A transition is marked terminal only when its next state has been empirically selected for expansion by the search procedure and the candidate-generation routine produces no accepted child under the same acceptance checks used by the released implementation. These checks include rule applicability, successful rewrite execution, executability under PostgreSQL EXPLAIN, positive EXPLAIN cost, and cost not exceeding the original query cost. Under this convention, a terminal flag

means that the next state has been tested and found non-expandable by the implemented rewrite procedure, not merely that the search stopped before reaching it.

Concretely, the data-collection wrapper records each generated transition with `done = false` at the moment the transition is created. It also maintains an in-memory map from each generated child-node identifier to the transition that produced that child. Later in the same data-collection run, if that child node is itself selected for further expansion by the original search procedure and the candidate-generation routine produces no accepted child from it under the existing acceptance checks, including rule applicability, executability under PostgreSQL EXPLAIN, positive cost, and cost not exceeding the original query cost, the stored transition is updated in place from `done = false` to `done = true`. Transitions whose child state is never selected for further expansion under the original search procedure, either because UCT directs exploration elsewhere or because the search budget is exhausted before that state is reached, retain `done = false`. The terminal flags written to the exported transition CSV at the end of the data-collection run therefore reflect the cumulative outcome of the entire run rather than a per-transition decision made at the time the transition is recorded.

The data-collection wrapper additionally includes an early-stopping mechanism in the search loop: when all current leaf nodes have been selected and none of them produces a new accepted child, the search terminates before spending the remaining budget. This avoids repeatedly attempting to expand an already-exhausted frontier and keeps the terminal-labeling convention consistent with the implemented notion of "no accepted child" rather than with elapsed budget. This convention deliberately favors conservative terminal labels: a transition is labeled `done = true` only after the next state has been tested and confirmed to produce no accepted child, which avoids falsely labeling budget-truncated states as terminal. The cost of this conservatism is that some transitions retain `done = false` even if their next state would have produced no child, because that next state was never selected for expansion before the search moved elsewhere or the budget was exhausted. The labeling can therefore contain false non-terminal cases, but it avoids false terminal cases caused solely by budget truncation. The implications of this asymmetry for the value learned by the model are discussed in Section 4.5.3.

Training data come from the released code itself, not from a separate data-generation run. The search proceeds normally. Whenever the framework successfully generates and accepts a candidate child, the data-collection wrapper records a transition. This way, the dataset reflects exactly the rewrite states the original search would visit. The recorded transitions are exported from the rewrite system to the training environment for offline DQN training. No online policy update or feedback loop is involved; once collected, the dataset is treated as a fixed offline supervision source.

3.3.2 DQN Model and Training Objective

The value model is implemented as a Q-network that maps the encoded state to action values over the rule-class action space. The input is the 32-dimensional feature vector defined in Section 3.2.1, and the output is a 7-dimensional vector of Q-values, one for

each rule class defined in Section 3.2.2. The network is a lightweight multilayer perceptron with two hidden layers, each with 128 units and ReLU activation. The final layer is a linear projection to the seven action values. The architecture is intentionally simple. The thesis does not aim to explore deep architectural design. It only asks whether a learned value model can guide expansion within an existing rewrite framework, and the simplest workable architecture is enough to answer that question.

Training follows the standard DQN setup introduced in Section 2.4.2. For each transition $(s_t, a_t, \tilde{r}_t, s_{t+1}, d_t)$ in a minibatch, the network produces predictions over all seven actions, and the value associated with the recorded action a_t is selected as the prediction. The target value combines the scaled reward $\tilde{r}_t$ defined in Equation 3.5 with the discounted maximum predicted value of the next state, computed by a separate target network:

$$y_t = \tilde{r}_t + \gamma(1 - d_t) \max_{a'} Q_{\text{target}}(s_{t+1}, a'), \tag{3.4}$$

where d_t is the terminal flag for transition t defined in Section 3.3.1 and γ is the discount factor. The discount factor is set to $\gamma = 0.99$ in the current implementation. The training objective is the mean squared error between the prediction $Q(s, a; \theta)$ and the target y. The maximum in Equation (3.4) is taken over the full seven-dimensional action space rather than over the legal-action subset of s'. This is a deliberate choice consistent with the recorded transition format, which stores the next state's encoded feature vector but not its legal-action mask. The consequence is that the bootstrap target at training time can in principle draw from action values that the inference-time legal-action mask would have suppressed at the corresponding state. The empirical effect of this train-inference asymmetry is discussed as a scope and validity consideration in Section 4.5.3. A separate target network is maintained for stable target computation. Its parameters are updated by soft updates with coefficient $\tau = 0.005$, applied at the end of each training epoch: at each update, the target network's parameters are moved slightly toward the online network's parameters via $\theta^- \leftarrow \tau\,\theta + (1-\tau)\,\theta^-$, not being copied directly. Optimization is performed using Adam with a learning rate of 10^{-3}. These choices follow standard DQN practice (Section 2.4.2) and are not modified for the present application.

Offline training is performed in a Python environment using the transition CSV exported from the rewrite system. Each row in the file corresponds to one collected transition. Before training, the raw reward r_t^{raw} recorded in the transition CSV is divided by a constant scaling factor $C_{\text{scale}} = 10^5$:

$$\tilde{r}_t = \frac{r_t^{\text{raw}}}{C_{\text{scale}}}. \tag{3.5}$$

The scaled reward $\tilde{r}_t$, rather than the raw EXPLAIN-cost difference, is used in the DQN target during optimization. This linear rescaling brings per-step cost-reduction magnitudes into a range where mean squared error against bootstrapped Q-targets remains numerically well-conditioned under the chosen learning rate. The rescaling preserves the relative ordering of rewards and the optimal policy implied by the training signal; the discount factor $\gamma = 0.99$ and the learning rate 10^{-3} were selected against the rescaled rewards rather than the raw EXPLAIN-cost differences. The dataset of recorded transitions is then split

into training and validation subsets using an 80/20 partition with a fixed random seed. The split is performed at the transition level rather than at the query level, so transitions from the same query may appear in both subsets. This split is internal to the 468-query training portion described in Section 4.1.5 and is used only as a signal for early-stopping model selection during training; the held-out 200-query test set described in Section 4.1.5 is never seen by the model in any form during training, validation, or model selection. A query-grouped split would more cleanly avoid intra-query correlation between the training and validation subsets and is left for future work.

Training proceeds in minibatches of size 256 for at most 100 epochs. Validation loss is monitored at the end of each epoch, and early stopping is triggered when validation loss does not improve beyond a small threshold for a fixed number of epochs. Rather than retaining the model from the final epoch by default, the training loop keeps the parameters corresponding to the best observed validation loss; this is the model used for downstream deployment. After training completes, the best-validated model is restored and exported to ONNX format for use outside the Python environment, as described in Section 3.4.

The standard DQN target in Equation (3.4) bootstraps from the model's own estimate of the next state's value. Under the conservative terminal-labeling convention described in Section 3.3.1, the bootstrap term is suppressed only when the next state has been empirically tested and found to produce no accepted child. It is not suppressed merely because the search budget ended before the next state was expanded. The bootstrap term therefore contributes to the training target on the majority of recorded transitions, propagating value information across the parts of the policy tree that were not exhaustively explored during data collection. The strength with which value information actually propagates through bootstrap is shaped jointly by the discount factor, by the labeling rule, and by the structure of the recorded transition data. Under the released search budget, the recorded trajectories are typically short, so the dataset rarely contains long chains of state transitions through which value could propagate, even though $\gamma = 0.99$ would in principle permit a longer effective horizon. The value information that the model learns therefore reflects near-horizon continuation rather than an arbitrarily long downstream rewrite. This is consistent with the role the model plays at expansion time, where it is asked to compare the seven rule classes at a single state rather than to plan over many steps ahead. It is also consistent with the limitations of the released implementation, which does not expose the selection-stage downstream-value signal described in the LearnedRewrite paper. Distinguishing the contribution of the bootstrap term from a pure supervised regression on transition-level reward would require an ablation that holds all other factors fixed; this is not performed in the present work and is identified as future work in Chapter 5.

The full set of DQN hyperparameters and architecture choices used to produce the model reported in Chapter 4 is summarized in Table 3.2. Two items conventionally listed for DQN training are deliberately omitted from the table because they do not apply under the present offline training setup:

- *Replay-buffer capacity.* The fixed transition CSV described in Section 3.3.1 constitutes the complete supervision source and is loaded into memory in its entirety during

Table 3.2: DQN architecture and training hyperparameters used in this thesis.

Component	Value
Network architecture	
Input dimension	32 (Section 3.2.1)
Hidden layers	Two fully connected layers
Hidden units per layer	128
Hidden activation	ReLU
Output dimension	7, one Q-value per rule class
Output activation	Linear
Weight initialization	PyTorch default
Training objective	
Loss function	Mean squared error against the target in Eq. (3.4)
Discount factor γ	0.99
Reward scaling C_{scale}	10^5 (Eq. 3.5)
Target network	Separate network initialized from the online-network weights
Target update rule	Soft update: $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$, where $\tau = 0.005$
Target update frequency	Once per training epoch
Optimization	
Optimizer	Adam with default $\beta_1 = 0.9$ and $\beta_2 = 0.999$
Learning rate	1×10^{-3}
Minibatch size	256
Minibatch sampling	Uniform random sampling without replacement within each epoch; reshuffled at the start of each epoch
Gradient clipping	None
Data and training schedule	
Train/validation split	80/20 transition-level split with fixed seed (`random_state`= 42)
Maximum epochs	100
Early-stopping monitor	Mean validation loss over validation transitions
Minimum improvement	1×10^{-4}
Early-stopping patience	10 epochs
Model selection	Parameters achieving the lowest observed validation loss
Actual stopping epoch	23 (Section 4.1.7)
Training wall-clock time	Approximately 18 minutes on the GPU described in Section 4.1.1

training; minibatches are sampled directly from this fixed pool, so no separate buffer with a finite eviction capacity is maintained.

- *Exploration schedule (e.g., ϵ-greedy).* No online interaction with the rewrite framework occurs during DQN training. The transitions used as supervision are produced once, by the unmodified released search procedure during the data-collection phase (Section 3.3.1), so the data-collection policy is independent of the Q-network being trained and no exploration schedule is required.

3.3.3 top-k Selective Expansion in Search

The trained value model is used only for inference during online query rewriting. Its role is limited to the expansion stage of the search loop, where it replaces the exhaustive enumeration of legal rule classes with a learned ranking and a bounded top-k subset. The rest of the search procedure remains unchanged: node selection, reward backup, and final best-node retrieval all operate as in the released implementation. The intervention is therefore local to expansion, not a redesign of the surrounding search.

Algorithm 1 specifies the DQRS expansion procedure at a selected node. Three structural points are worth noting: the Q-network output is masked to suppress illegal rule classes before ranking; the effective top-k is reduced when fewer than k rule classes are legal at the current state; and a selected rule class only defines a rewrite attempt, while child acceptance still depends on the unchanged executability and cost checks from the released implementation.

Algorithm 1 DQRS top-k expansion at a selected node

Require: Selected node v; original query state s_0; top-k value k; trained Q-network Q_θ
Ensure: Accepted child nodes inserted under v

$M \leftarrow \textsc{LegalMask}(\mathrm{Rel}(v))$
$x \leftarrow \textsc{Encode}(\mathrm{Rel}(v), C_{\mathrm{EXPLAIN}}(q(v)))$
$\mathbf{q} \leftarrow \textsc{Mask}(Q_\theta(x), M)$
$A \leftarrow \textsc{TopK}(\mathbf{q}, \min(k, \|M\|_0))$
for all $a \in A$ **do**
    $(s', q(s'), \mathrm{changed}) \leftarrow \textsc{TryRewriteByRuleClass}(\mathrm{Rel}(v), a)$
    if $\mathrm{changed} \wedge \textsc{SafeExplain}(q(s')) \wedge 0 < C_{\mathrm{EXPLAIN}}(s') \leq C_{\mathrm{EXPLAIN}}(s_0)$ **then**
        Insert s' as a child of v
    end if
end for

In the implementation, TryRewriteByRuleClass corresponds to the single-rule rewrite routine called with one selected rule class. It configures the HepPlanner with the concrete rules registered under that class, returns a rewritten candidate with changed = true when at least one concrete rule fires, and returns changed = false otherwise.

The following numbers are illustrative and are used only to show the mechanics of masking and top-k selection. Suppose the action indices are mapped to rule classes as follows:

$$1:\texttt{rule\_filter},\quad 2:\texttt{rule\_join},\quad 3:\texttt{rule\_agg},\quad 4:\texttt{rule\_project},$$
$$5:\texttt{rule\_cal},\quad 6:\texttt{rule\_orderby},\quad 7:\texttt{rule\_union}.$$

At a selected state v, assume the relational tree contains filter, join, and project patterns but no aggregate, calc, sort, or union pattern. The legal-action mask is therefore

$$M = [\,1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0\,].$$

Suppose the Q-network returns

$$\mathbf{q} = [\,0.355,\ 0.674,\ 0.350,\ 0.448,\ -0.182,\ -0.033,\ -0.218\,].$$

After masking, the values used for ranking become

$$\mathbf{q}_{\text{masked}} = [\,0.355,\ 0.674,\ -10^9,\ 0.448,\ -10^9,\ -10^9,\ -10^9\,].$$

With $k = 3$, the effective value is $k_{\text{eff}} = 3$, and the selected rule classes in ranked order are

$$[\,\texttt{rule\_join},\ \texttt{rule\_project},\ \texttt{rule\_filter}\,].$$

The rewrite routine then attempts these three rule classes in that ranked order. If `rule_join` produces a PostgreSQL-safe child with positive EXPLAIN cost no larger than the original query cost, that child is inserted into the policy tree. If `rule_project` produces a rewritten SQL string that fails the PostgreSQL EXPLAIN safety check, it is discarded. If `rule_filter` does not fire any concrete rule, that is, if the rewrite routine returns changed = false, no child is inserted for that action. Thus top-k controls which rule classes are *attempted*, while the existing rewrite and filtering logic still determines which attempted classes actually become children.

Figure 3.2 gives the corresponding system-level view, summarizing the same flow as Algorithm 1 in diagram form, with the key separation between the Q-network ranking step and the framework's child-acceptance logic visually highlighted.

The proposed mechanism therefore changes which candidate rule classes are expanded at each step, but not how accepted child nodes participate in the rest of the search. Within this design, k acts as an explicit control over pruning aggressiveness: a smaller k enforces more selective expansion and reduces branching more strongly, while a larger k preserves more candidate diversity at the cost of additional search effort. The value of k is a configurable parameter of DQRS rather than a fixed hyperparameter; its effect is examined empirically in Chapter 4.

3.3.4 Design Rationale and Computational Trade-Off

The motivation for DQRS follows from the limitations identified in Chapter 1, with the scope condition stated there: this thesis addresses the state-agnostic nature of existing pruning, but does not address the absence of an explicit downstream-value signal in the released implementation. Per-candidate gating in the released implementation treats each rule class in isolation and provides no mechanism for ranking the rule classes available at the current state against one another. A state-dependent learned ranker addresses this gap directly: given features summarizing the current state, the network produces scores over the seven rule classes that can be used to order them. The Q-learning training objective in Equation (3.4) provides one way to derive these scores from observed rewrite transitions, with the bootstrap term contributing short-horizon continuation information under the data-collection setup described in Section 3.3.1. Stronger forms of long-horizon reasoning would require either a different training procedure with longer recorded trajectories, or, more naturally, the restoration of the paper's learned cost estimator as a source of downstream-value signal; neither extension is pursued in this thesis.

Top-k generalizes the ranking decision: $k = 1$ corresponds to fully greedy single-child expansion, while larger k preserves more branching diversity at the cost of more candidates

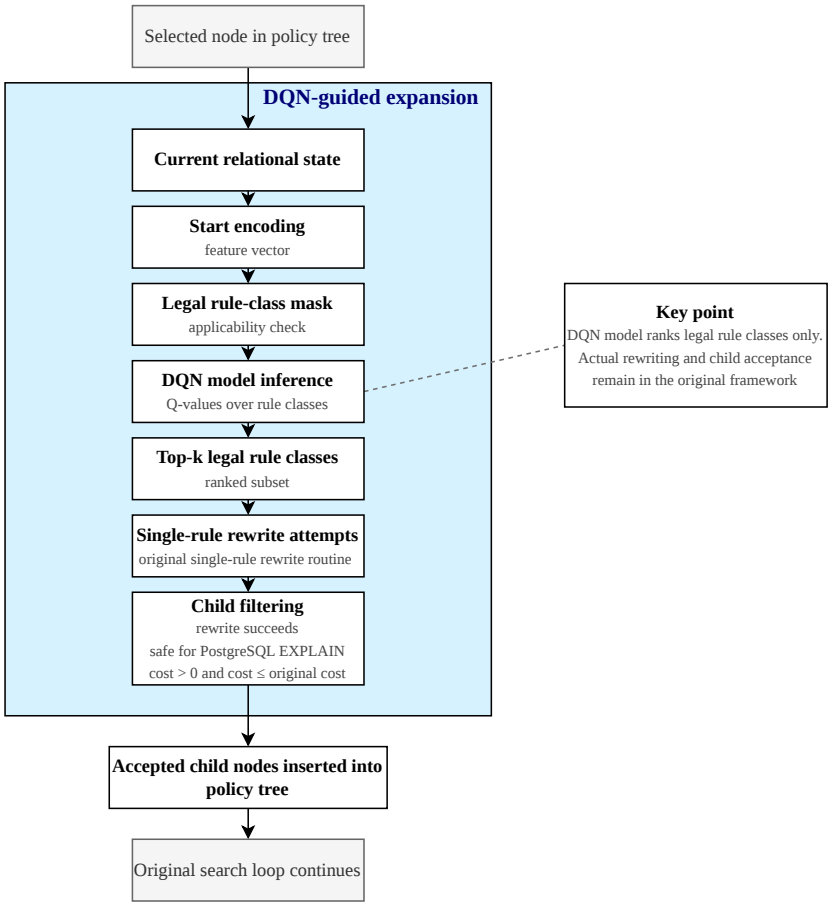

Figure 3.2: DQRS-guided top-k expansion at a selected search node.

per node. Both regimes are admissible under the same masking and ranking machinery, and the appropriate value is workload-dependent rather than fixable a priori. The choice of k therefore reflects a trade-off between branching diversity and search speed. The empirical effect of this knob is examined in Section 4.3.1, which shows that the appropriate setting differs between the in-domain TPC-H-derived workload ($k = 1$) and the cross-workload TPC-DS-derived workload ($k = 4$).

The computational trade-off implied by DQRS is small inference cost in exchange for reduced expansion overhead. Per-state encoding, mask construction, and Q-network forward pass together add a small overhead at each expanded node. Against this, every rule class that DQRS prunes saves the cost of one full candidate-generation cycle in the released implementation, which includes rule application, executability check, and EXPLAIN-based cost evaluation. The relative magnitudes of these two costs determine whether the trade-off is favorable in practice. The empirical evaluation in Chapter 4 measures this trade-off directly through rewrite latency and node count. Whether the DQN training apparatus in this setup contributes substantively beyond what a supervised regression on transition-

level reward would yield is an open question identified as future work in Chapter 5. The "DQN-guided" naming reflects the training procedure used in this thesis, not an empirically established advantage of DQN over alternative value-fitting procedures.

3.4 Implementation of DQRS

DQRS is implemented as a two-environment system. The rewrite framework, including the search loop, candidate generation, EXPLAIN-based cost evaluation, and policy-tree maintenance, runs in Java on top of the released LearnedRewrite single-rule codebase. Q-network training runs in Python with PyTorch in a separate environment. The two environments are linked by two artifacts: a transition CSV exported from the Java side for training, and an ONNX model exported from the Python side for inference. There is no live connection between the two during evaluation; the trained model is treated as a fixed inference component once exported.

Inside the Java rewrite framework, each policy-tree node carries the rewrite state in two forms: an executable SQL string and the corresponding Calcite relational tree (`RelNode`). Rule applicability and rewrite execution both consume the `RelNode` form; the SQL string is retained only for PostgreSQL EXPLAIN cost evaluation and for producing the final rewritten query at the end of search. DQRS reuses this two-form representation unchanged. At each search node where the value model is invoked, the encoder reads the `RelNode` together with the most recent EXPLAIN cost of the corresponding SQL form and produces the 32-dimensional feature vector defined in Section 3.2.1; the feature vector, not the `RelNode` or the SQL string, is what is passed to the Q-network and what is recorded in the transition tuple.

On the Java side, transition collection is implemented by adding a data-collection variant of the child-generation routine in the rewrite system. Whenever a candidate child is successfully generated and passes the existing acceptance checks, the routine records a transition consisting of the encoded parent state, the rule class used, the observed cost-based reward, the encoded child state, and the terminal flag, with the terminal flag initially set to false. The data-collection wrapper additionally maintains an in-memory index from each recorded child state to the corresponding transition record so that the terminal flag can be retroactively updated when the child state is later processed and produces no valid grandchild, as described in Section 3.3.1. The records are accumulated across a training-set workload and written to a transition CSV at the end of the run, at which point the terminal flag of each record reflects its final post-run value. The rest of the search behavior during data collection is identical to the released implementation, so the resulting dataset reflects the actual distribution of rewrite states and transitions seen by the original system. This design required a data-collection variant of the child-generation routine rather than simply logging the final frontier when the search budget is exhausted. The purpose of this variant is to distinguish true rewrite terminality from search truncation caused by the finite MCTS budget. Under the terminal-labeling convention introduced in Section 3.3.1, a state is treated as terminal only when it is selected for expansion and produces no accepted child.

The Python training environment loads the exported transition CSV, applies the reward scaling described in Section 3.3.2, performs the 80/20 train-validation split with a fixed random seed, and trains the Q-network for at most 100 epochs with early stopping on validation loss. Training uses Adam with learning rate 10^{-3} and minibatch size 256, with the soft target update controlled by $\tau = 0.005$ and discount factor $\gamma = 0.99$. The best-validated model parameters are retained as the final model.

After training, the best-validated model is exported to ONNX format. The export uses a CPU copy of the trained network and a fixed dummy input matching the 32-dimensional state representation. On the Java side, the exported ONNX model is loaded through the ONNX Runtime Java API and used purely for inference during query rewriting. No additional training, parameter update, or feedback loop is performed in the deployed system. This separation keeps the rewrite framework independent from the Python environment while still allowing the trained ranker to be invoked at every expansion step at the cost of a single forward pass.

At runtime, DQRS exposes a small number of control parameters. The most important is the top-k value, which controls pruning aggressiveness and is chosen empirically per workload. Other parameters, such as the rewrite budget and the rule-class set, are inherited from the released implementation and not modified. Detailed hardware configurations, software versions, and the exact parameter values used in the reported experiments are described in Chapter 4. The remainder of this thesis evaluates the resulting system empirically against the released baseline.

Chapter 4

Experimental Study

This chapter evaluates DQRS within the released LearnedRewrite single-rule implementation. The evaluation focuses on three properties: rewrite quality, rewrite overhead, and the size of the search space explored during rewriting. Because DQRS modifies the expansion stage rather than replacing the surrounding rewrite framework, the natural comparison target is the released implementation itself, run under identical conditions. Section 4.1 describes the experimental environment, the workloads, the metrics, and how training data are collected. Section 4.2 reports the main results on the TPC-H-derived and TPC-DS-derived workloads. Section 4.3 examines the effect of top-k and compares DQN-guided ranking against random ranking under the same masking constraint. Section 4.4 interprets the patterns observed across the experiments. Section 4.5 closes the chapter with a discussion of the scope and validity of the empirical findings.

4.1 Experiment Setup

The task evaluated in this chapter is the following. Given an input SQL query, produce a semantically equivalent rewritten query by searching over sequences of rule-class applications, under a fixed rewrite budget, using the released LearnedRewrite single-rule framework. The released implementation performs this task by enumerating every legally applicable rule class at each expanded node. DQRS performs the same task under the same framework, the same rule set, the same acceptance conditions and the same budget, differing only in that expansion forwards at most k rule classes selected by the learned ranker instead of all legal ones. What is measured is therefore not whether rewriting is possible but what it costs to arrive at a rewrite of comparable estimated quality: the experiments compare the two methods on the estimated quality of the rewritten query, the wall-clock cost of producing it, and the number of states accepted into the search tree along the way.

This section describes the setup used in the experiments. The structure follows the order in which a reader would need the information: hardware and software environment first, then datasets, baseline, metrics, data partitioning, transition collection for training, and training time.

4.1.1 Environment

All experiments ran on a machine with a Windows 11 Home host (version 25H2), an Intel Core Ultra 7 265F CPU, and 64 GB of memory. The evaluation and integration code was written in Java 25.0.1. The database backend was PostgreSQL 16.11, deployed inside a Docker container based on the official `postgres:16` image. The container was run without explicit CPU or memory limits, so the database process had access to the full resources of the host described above. No other workload was running on the host during measurement. Because the same environment is used for the released baseline and for DQRS, and each query is measured three times with the average taken, this affects the absolute latency values more than the comparison between methods. A tightly controlled resource configuration would be required for the end-to-end execution-time study identified as future work in Section 5.2. DQN training ran in a separate Docker container with Python 3.11.13 and PyTorch 2.8.0+cu128, with CUDA 12.8 available; the training environment had access to an NVIDIA GeForce RTX 5070 GPU. After training, the model was exported to ONNX and loaded back into the Java side through ONNX Runtime 1.18.0 for inference during query rewriting, running on CPU.

Because every cost value used in this thesis is obtained from PostgreSQL EXPLAIN, the planner configuration under which those estimates are produced is part of the experimental setup. Each database connection applies a fixed set of session-level settings before any query is evaluated. Both `join_collapse_limit` and `from_collapse_limit` are set to 1. The first prevents the planner from flattening explicit `JOIN` constructs into the surrounding `FROM` list, so the nesting expressed by explicit join syntax is preserved rather than being reordered; the second prevents subqueries in the `FROM` list from being merged into the parent query. The effect on join ordering therefore depends on how a query is written, and is strongest for queries that use explicit join syntax. Genetic query optimization is disabled by setting `geqo` to off. The three join methods, hash join, merge join and nested-loop join, are left enabled. These settings are identical for the released baseline and for DQRS, and they are applied once per connection rather than per query.

Two consequences follow. First, constraining how far the planner may restructure the submitted query keeps the effect of a logical-layer rewrite visible in the cost estimate. Where the planner is free to reshape the query itself, part of the structural change produced by a rewrite rule can be absorbed by the planner's own search, and the cost signal understates the difference between a state and its rewritten successor. Second, disabling genetic query optimization removes a stochastic component from the planner: `geqo` uses a randomized search for queries with many joins, so leaving it enabled could produce different cost estimates for the same query across evaluations. Together these settings are what allow the cost-reduction and node-number metrics to be described as deterministic in Section 4.1.4. The implementation extends the released LearnedRewrite single-rule branch [29], with the expansion stage replaced as described in Chapter 3.

4.1.2 Dataset

The in-domain workload is derived from TPC-H, a decision-support benchmark published by the Transaction Processing Performance Council (TPC) that models a wholesale supplier and consists of ad-hoc analytical queries over eight tables. The cross-workload evaluation uses TPC-DS, a later benchmark from the same body that models a retail decision-support environment with a substantially larger schema and a broader mix of query shapes.

TPC-H-derived workload. The starting point is the synthetic TPC-H-like query pool released alongside the public LearnedRewrite codebase [30], which contains 10,000 queries generated to exercise the rewrite rules exposed by the released single-rule implementation. This pool is not the standard set of 22 TPC-H benchmark queries; it is the workload aligned with the released implementation used as the practical baseline in this thesis. Because the released code uses PostgreSQL EXPLAIN cost as its quality signal, the pool was re-evaluated under the current environment, and only queries on which the released single-rule rewrite procedure produces a rewritten query whose estimated PostgreSQL EXPLAIN cost is strictly less than the original query cost were retained. The resulting release-aligned subset contains 668 queries, or 6.7% of the original 10,000, and is stored in `tpch_668.txt`.

The 668 queries are split at the query level. `tpch_dqn_train_468.txt` contains 468 queries used to collect rewrite transitions for offline DQN training. `tpch_test_200.txt` contains 200 held-out queries used for the main TPC-H-derived test results. This corresponds to a 70/30 split of the release-aligned subset, performed once with a fixed random seed and held constant across all reported experiments. Throughout this chapter, the TPC-H-derived results refer to the 200-query held-out test set, not to the full 668-query source pool.

TPC-DS-derived workload. The starting point is a pool of 1,089 queries generated from the official TPC-DS query templates using the standard `dsqgen` tool. The toolkit was obtained from the publicly available `gregrahn/tpcds-kit` repository [19], a community-maintained fork of the official TPC-DS v2.10.0 toolkit that adds macOS compilation support and applies documented fixes to template bugs in `query22a`, `query77a`, and the `s_web_returns` column naming, while leaving the `dsqgen` tool and the underlying query templates otherwise identical to the official v2.10.0 release. Queries were generated across ten independent query streams, from stream 0 through stream 9, yielding 99 queries per stream and 990 queries in total, supplemented to 1,089 queries for use in this thesis. The 1,089 generated queries were passed through the same cost-based filter applied to the TPC-H-derived workload: only queries on which the released single-rule rewrite procedure produces a rewritten query whose estimated PostgreSQL EXPLAIN cost is strictly less than the original query cost were retained. Additional minor adjustments were made for compatibility with the PostgreSQL/Calcite-based rewriting pipeline, including manual corrections where needed. The final subset contains 123 queries, or 11.3% of the original 1,089-query pool, and is stored in `tpcds_test_123.txt`.

The TPC-DS-derived workload is used only as a cross-workload validation set, not for DQN training. The reason is the scale of the filtered pool: 123 queries are not sufficient to support a separate train/test split that would allow an in-domain TPC-DS-trained model to be compared against an in-domain TPC-DS-tested baseline at the same statistical resolution used for the TPC-H-derived workload. Consequently, the model evaluated on the TPC-DS-derived workload throughout this chapter is the TPC-H-trained model, and the TPC-DS-derived results reflect cross-workload behavior rather than in-domain training. Constructing a larger TPC-DS-derived pool that would support an in-domain training/test split is identified as future work in Section 5.2.

The rewrite framework operates over the seven rule classes defined in Section 3.2.2. Unless stated otherwise, the rewrite budget is fixed at 20, which matches the default of the released single-rule implementation. In the released code, this budget bounds the number of MCTS iterations executed per query, and a single iteration can add multiple nodes to the tree during expansion. The DQN-guided expansion uses top-$k = 1$ on the TPC-H-derived test set and top-$k = 4$ on the TPC-DS-derived validation workload. These workload-specific settings are chosen based on the ablation in Section 4.3. Because these workload-specific top-k values are selected using the same evaluation workloads reported in the ablation study, the corresponding main-result rows should be interpreted as operating-point characterizations rather than as estimates under a separately tuned validation protocol.

4.1.3 Baseline

The baseline used in all experiments is the publicly released single-rule implementation of LearnedRewrite. This implementation realizes the policy-tree search structure described in the LearnedRewrite paper but does not include the paper's learned cost estimator (Section 3.1.3); cost evaluation is provided by PostgreSQL EXPLAIN. All comparisons in this chapter are therefore between DQRS and this released implementation, not between DQRS and the full system proposed in the paper. To reduce ambiguity, this baseline is referred to throughout as the released implementation rather than as "the LearnedRewrite baseline". In the released implementation, the expansion stage enumerates all seven rule classes when generating candidates at a selected node. DQRS keeps the rest of the framework unchanged, including the same rewrite system, search procedure, and rule set, and replaces only this exhaustive enumeration with DQN-guided top-k selection. The rule engine itself is independent of the technique proposed in this thesis. Rewriting is carried out by Apache Calcite's `HepPlanner`, which is configured with the concrete rules of a selected class and invoked through the released implementation's existing rewrite routine. DQRS does not modify the planner, the rules registered under each class, the order in which the planner applies them within a class, or the conditions under which a produced child is accepted. It determines only which rule classes are handed to that routine. The same planner and the same rule inventory are therefore used by both methods. The comparison therefore isolates the effect of DQN-guided rule-class selection within the same rewrite framework. A single primary baseline is appropriate here because DQRS is a module-level modification of the surrounding system. Both methods share the same codebase, database environment, rewrite rules, and workloads. Differences in cost reduction, latency, and node count can

therefore be attributed to the proposed DQN-guided expansion rather than to broader system-level differences.

4.1.4 Metrics

The first metric is *estimated cost reduction*, reported in the tables as cost reduction. For an input query q, let q_{out} be the final query returned by a rewrite method, and let $C_{\text{EXPLAIN}}(\cdot)$ denote the PostgreSQL EXPLAIN total-cost estimate defined in Section 3.3.1, applied here to the original query and to the final rewritten query produced by a rewrite method. The estimated cost reduction is

$$\Delta C(q) = C_{\text{EXPLAIN}}(q) - C_{\text{EXPLAIN}}(q_{\text{out}}). \tag{4.1}$$

A larger $\Delta C(q)$ means that the rewritten query has a lower optimizer-estimated cost relative to the original query. This metric is an estimator-side proxy for rewrite quality, not a measurement of true execution-time improvement. It is used here because the released implementation itself uses PostgreSQL EXPLAIN cost as its internal quality signal. Within this chapter, cost reduction is therefore used to evaluate whether DQRS limits degradation in the released baseline's estimator-side cost-reduction signal while reducing rewrite overhead, rather than to claim strict runtime improvement.

The second metric is *rewrite latency*, denoted T_{rewrite}, measured in milliseconds. It is the wall-clock time spent inside the rewrite-search routine after the original relational representation and original EXPLAIN cost have been obtained. For the released baseline, this includes MCTS search, candidate generation, rewrite attempts, child executability checks, and PostgreSQL EXPLAIN evaluations for candidate children. For DQRS, it additionally includes state encoding, legal-action masking, ONNX-based Q-network inference, and top-k ranking during expansion. It excludes offline DQN training, execution time of the final rewritten query, and the initial SQL-to-relational-tree conversion. This metric captures the overhead of producing a rewritten query, not the runtime of executing that query.

The third metric is *node number*, denoted N_{tree}, defined as the number of nodes in the policy tree after the rewrite search terminates, including the root node corresponding to the original query. Each non-root node corresponds to an accepted rewritten state inserted into the tree. Therefore, node number measures the size of the accepted search tree, not the total number of attempted rule-class applications: rewrite attempts that fail to fire, fail PostgreSQL EXPLAIN safety checks, or are rejected by the cost filter do not become tree nodes. A smaller node number indicates that the method explored fewer accepted candidate states. This metric is used as a structural proxy for search effort, complementary to rewrite latency.

For all three metrics, the reported summary statistics include the average, the median, and the population standard deviation. For rewrite latency, p90 is also reported. The p90 value is the 90th percentile: 90% of the per-query latency measurements fall at or below this value. It is included because rewrite latency can have a long upper tail, and the median alone may hide slow outlier rewrites. For cost reduction and node number, the average,

median, and standard deviation are used to describe the level and variability of the results. For the released baseline and DQRS, cost reduction and node number are deterministic given the same query, search procedure, and database state. For the random-k ablation, both metrics are deterministic given the fixed random seed; seed variability is discussed in Section 4.5.4. Rewrite latency is sensitive to short-term system noise. Therefore, each query is run three times under each method, and the average of the three runs is taken as the per-query latency. Statistics are then computed across all queries in the corresponding workload.

All quantities in this chapter are reported at a precision determined by the dispersion of the underlying measurements rather than by the precision at which they were recorded. The cost-reduction distributions on both workloads are strongly dispersed: the standard deviation exceeds the mean by a factor of about two on the TPC-H-derived workload and by a factor of about four on the TPC-DS-derived workload. Reporting cost-reduction values to their full recorded precision would therefore imply a resolution the measurements do not support, and would invite comparisons between values whose difference is several orders of magnitude below the spread of the data.

Cost-reduction values are consequently reported to three significant figures. Rewrite-latency values are reported to three significant figures, except that values below 10 ms are given to one decimal place, where the difference between methods is comparable to the reported precision. Node-number statistics are reported to one decimal place. Derived percentages are given to at most two significant figures.

Two consequences of this convention should be noted. First, values that differ only beyond the reported precision appear identical in the tables; Section 4.5.5 states which differences this thesis treats as resolvable and which it does not. Second, where two settings are stated to produce identical cost reduction, the identity holds in the underlying unrounded measurements and is not an artifact of rounding.

4.1.5 Data Partitioning

The TPC-H-derived workload is partitioned at the query level. 468 queries (`tpch_dqn_train_468.txt`) form the training portion, used to collect rewrite transitions for offline DQN training. The remaining 200 queries (`tpch_test_200.txt`) form the held-out test portion, used for the main TPC-H-derived results. The split was performed once with a fixed seed and is held constant across all reported experiments.

The TPC-DS-derived workload is not used for DQN training. All 123 queries in `tpcds_test_123.txt` serve as a separate validation workload. The TPC-DS-derived results in this chapter therefore reflect the behavior of a TPC-H-trained model evaluated on a different workload distribution. They do not reflect performance under in-domain training.

4.1.6 Training Sample Collection

The DQN model used in this thesis was trained only on transitions collected from the 468-query TPC-H-derived training split. As described in Section 3.3.1, transitions are collected

by running the original search procedure on the training queries and instrumenting the child-generation step. The overall search proceeds through the standard search entry point of the released implementation, but the child-generation routine is replaced with a data-collection variant that records each generated transition.

Each recorded transition contains the encoded current state, the selected rule class as the action, the observed cost-based reward, the encoded next state, and a terminal flag, together with query- and node-level identifiers used for bookkeeping. Recorded transitions are exported from the Java side as a CSV file and used in the Python training environment as the offline supervision source. The training procedure itself is described in Section 3.3.2 and the deployment path in Section 3.4; it is not repeated here.

4.1.7 Training Time

Offline DQN training was performed on the recorded transition CSV using the procedure in Section 3.3.2. Training ran for at most 100 epochs with minibatch size 256, with early stopping triggered when validation loss stopped improving by more than a small threshold. In the run used to produce the reported model, training converged at epoch 23, taking approximately 18 minutes on the GPU described in Section 4.1.1, and the parameters with the best validation loss were retained as the final model. Training time was a one-time offline cost and is not included in the rewrite latency reported in Section 4.2.

4.2 Main Results

This section reports the main results of DQRS against the released baseline. Two workloads are evaluated separately: the held-out TPC-H-derived test set in Section 4.2.1, and the TPC-DS-derived validation workload in Section 4.2.2. The TPC-H-derived workload is the in-domain setting, since the DQN model was trained on transitions from the corresponding training split. The TPC-DS-derived workload is the cross-workload setting and tests whether the learned rule-class preferences transfer beyond the training distribution. The effect of specific design choices, such as the top-k value, is examined separately in Section 4.3.

4.2.1 Results on the TPC-H-derived workload

Table 4.1 reports the main results on the held-out TPC-H-derived test set. DQRS uses top-$k = 1$ on this workload, as motivated in Section 4.3.1.

On cost reduction, DQRS retains the workload average of the released implementation: the two coincide at the reported precision, both shown as 4230 in Table 4.1, and differ by about 0.1% in the unrounded measurements. The median, however, drops from 156 to 103, a reduction of approximately 34%. The two summary statistics disagree because the cost-reduction distribution is heavy-tailed: the standard deviation, approximately 9790,

Table 4.1: Main results on the TPC-H-derived workload. Bold values indicate the best result for each primary metric. Values that coincide at the reported precision, and differences that Section 4.5.5 treats as unresolvable, are not bolded.

Method	Cost Reduction ↑			Rewrite Latency (ms) ↓				Node Number ↓		
	Avg	Median	Std	Avg	Median	P90	Std	Avg	Median	Std
LearnedRewrite released	4230	**156**	9790	15.3	4.6	42.9	21.1	9.9	4.0	10.3
DQRS	4230	103	9790	**4.0**	**3.5**	**5.3**	3.0	**2.4**	**2.0**	0.5

is more than twice the average, indicating that a small number of high-impact rewrites dominate the mean. In this regime the median is the more reliable description of typical-case cost reduction, and the typical-case cost reduction is meaningfully lower under DQRS than under the released implementation. This is the efficiency–quality trade-off implicit in aggressive top-k pruning, and it should be read as a property of the method at top-$k = 1$ rather than as noise.

On rewrite latency and search effort, the change is substantial. Average latency drops from 15.3 ms to 4.0 ms, a reduction of about 74%. The median drops from 4.6 ms to 3.5 ms, and p90 drops from 42.9 ms to 5.3 ms. The standard deviation falls from 21.1 ms to 3.0 ms, indicating that DQRS reduces both the level of latency and its variability across queries. Average node number drops from 9.9 to 2.4, a reduction of about 76%, with the median dropping from 4 to 2. Taken together, these numbers show that DQRS substantially reduces rewrite-search overhead on this workload.

Figure 4.1 shows a per-query comparison of rewrite latency. Each point is one test query; the diagonal line indicates equal latency between the two methods. Most points lie below the diagonal: DQRS is faster on 128 of 200 evaluated queries and slower on 72. The latency improvement is therefore broad in aggregate but is not universal at the per-query level.

Figure 4.2 shows the distribution of node-number differences (DQRS minus released baseline). The distribution is shifted strongly toward negative values. DQRS produces fewer policy-tree nodes on 196 of 200 evaluated queries and the same number of nodes on the remaining 4. It does not generate more nodes than the released baseline on any evaluated query in this run. The latency reduction is therefore accompanied by a broad and consistent reduction in search effort.

The summary statistics in Table 4.1 understate how differently the two methods behave across queries. Under the released implementation the per-query node count is not unimodal. It ranges from 3 to 29, but no query produces a tree of between 5 and 25 nodes: 150 of the 200 test queries produce trees of three or four nodes, and the remaining 50 produce trees of 26 to 29 nodes. Expansion adds at most three children at any node in this workload, as observed in the recorded transitions of Section 4.1.6, so a query either exhausts its applicable rewrites within the first few iterations or continues until the rewrite budget of 20 iterations is reached. There is no intermediate regime, and policy-tree size in this setting is governed by the budget rather than by any structural property of the tree itself.

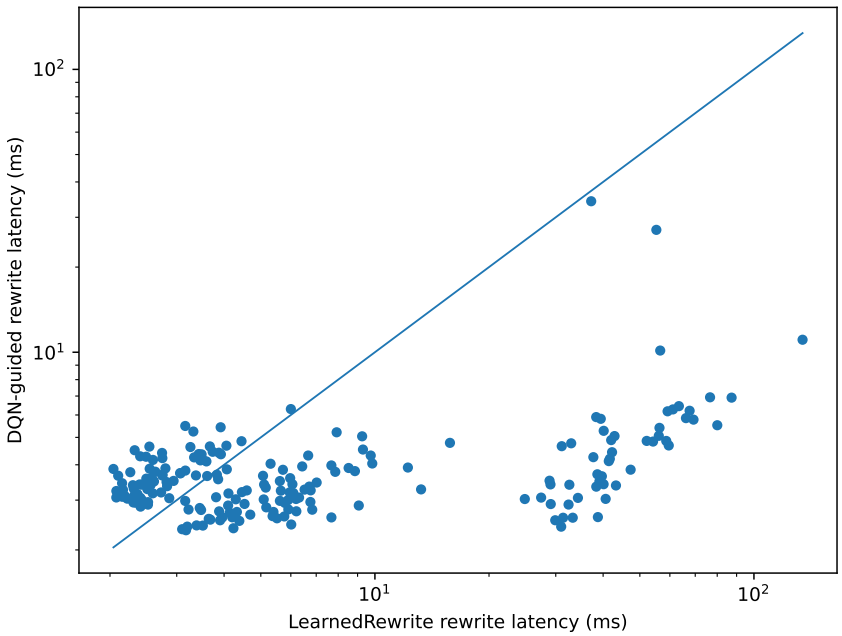

Figure 4.1: Per-query rewrite latency on the TPC-H-derived workload.

This shape explains where the reduction in search effort originates. DQRS does not shrink every tree by a similar factor; it removes the upper mode entirely. Under DQRS at k = 1 the node count takes only three distinct values across the test set, with 121 queries producing exactly two nodes and 78 producing three, and the standard deviation falls from 10.3 to 0.5. The question of whether a separate preprocessing or pruning stage would be required in practice is therefore answered in the affirmative for the released implementation on the upper mode, and DQRS is itself that pruning mechanism. It acts at expansion, before nodes are created, rather than as a post-hoc reduction of a tree that has already been built.

Taken together with the cost-reduction observations above, the TPC-H-derived results show that DQRS reduces rewrite-search overhead substantially while keeping average cost reduction close to the released baseline, at the cost of a lower typical-case cost reduction. The reported numbers reflect DQRS at $k = 1$; the choice of operating point is consequential here. As shown later in Section 4.3.1 (Table 4.3), increasing k to 2 on the same workload restores the median cost reduction to the released baseline level (156 in both cases), but average rewrite latency rises from 4.0 ms to 28.9 ms, exceeding the baseline's 15.3 ms by approximately 88%. The TPC-H results should therefore be read as evidence that DQRS exposes an explicit efficiency–quality trade-off rather than as evidence that DQRS dominates the released baseline at any single operating point. Section 4.4 returns to this characterization after the random-k ablation has been introduced.

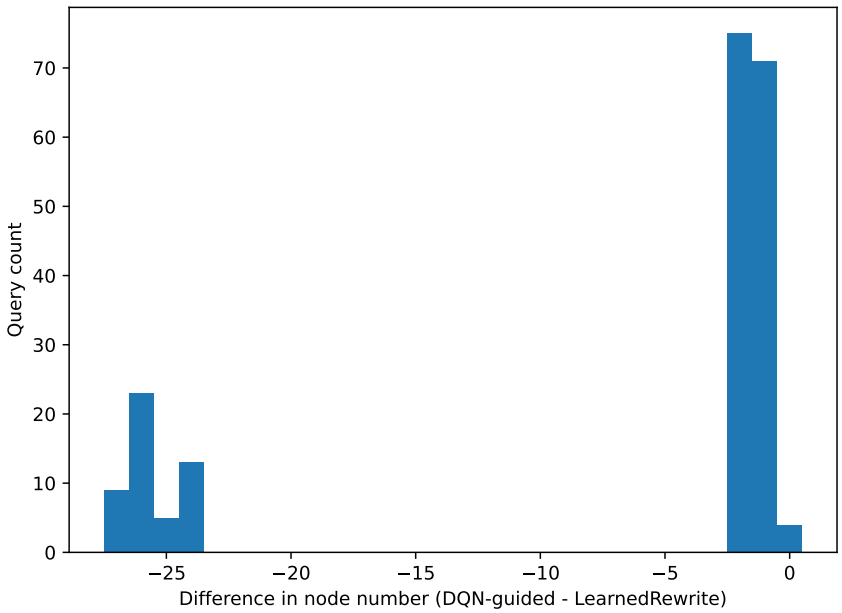

Figure 4.2: Distribution of per-query node-number differences on the TPC-H-derived workload.

Table 4.2: Main results on the TPC-DS-derived workload. Bold values indicate the best result for each primary metric. Values that coincide at the reported precision, and differences that Section 4.5.5 treats as unresolvable, are not bolded.

Method	**Cost Reduction ↑**			**Rewrite Latency (ms) ↓**				**Node Number ↓**		
	Avg	**Median**	**Std**	**Avg**	**Median**	**P90**	**Std**	**Avg**	**Median**	**Std**
LearnedRewrite released	96800	1200	377000	4440	1840	13000	6250	24.3	15.0	20.4
DQRS	96600	1200	377000	**3360**	**1110**	**9000**	5010	**17.1**	15.0	13.5

4.2.2 Results on the TPC-DS-derived workload

Table 4.2 reports the main results on the TPC-DS-derived validation workload. DQRS uses top-$k = 4$ on this workload, with the choice motivated by the ablation in Section 4.3.1.

On this workload, DQRS preserves cost reduction more closely than on the TPC-H-derived test set. Average cost reduction is 96600 versus 96800 for the released baseline, a difference of about 0.1%, and the median is identical at 1200 for both methods. Under the top-$k = 4$ setting used here, the value model retains a wider candidate set than under top-$k = 1$, which allows it to keep up with the released baseline on cost reduction.

On rewrite latency, DQRS reduces average latency from 4440 ms to 3360 ms, a reduction of about 24%. The median drops from 1840 ms to 1110 ms and p90 drops from 13000 ms to 9000 ms. Average node number drops from 24.3 to 17.1, a reduction of about 30%, while the median is unchanged at 15. The reductions are clearly visible but smaller in magnitude than on the TPC-H-derived test set, which is consistent with two factors: the model is evaluated under a workload it was not trained on, and the top-k setting used here is less aggressive.

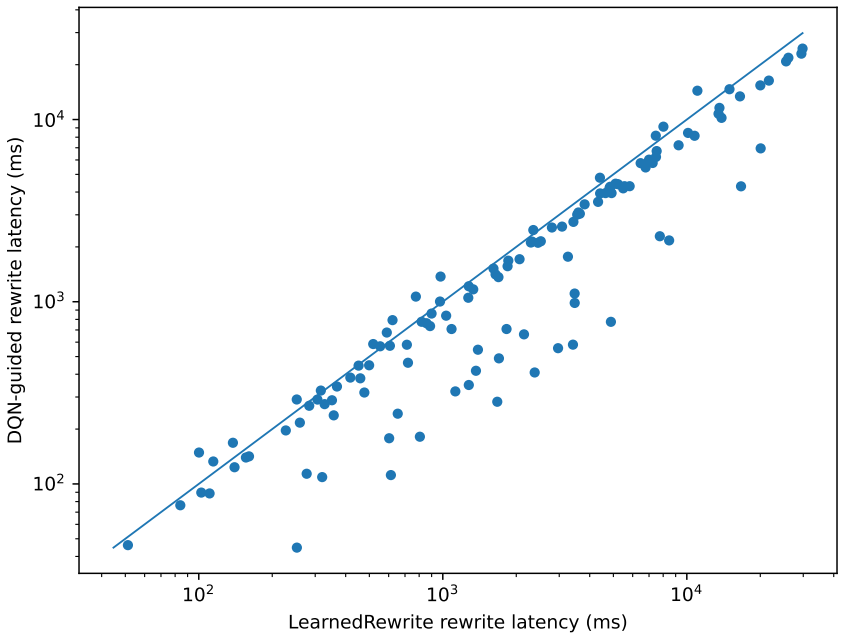

Figure 4.3: Per-query rewrite latency on the TPC-DS-derived workload.

Figure 4.3 shows per-query rewrite latency. Most points sit below the diagonal: DQRS is faster on 106 of 123 queries and slower on 17. The latency improvement is broad but, as on the TPC-H-derived workload, not universal.

Figure 4.4 shows the distribution of node-number differences. Most values are negative or zero: DQRS produces fewer nodes on 77 queries and the same number on 44. There are also two queries on which DQRS produces slightly more nodes. The result is therefore not strict per-query dominance on this workload; it is a clear overall shift toward fewer generated nodes, with a small number of exceptions. Compared with the in-domain TPC-H-derived results, the magnitude of efficiency improvement on TPC-DS is smaller, while the cost reduction profile is closer to that of the released implementation. The interpretation of these results requires the random-selection comparison reported in Section 4.3.2, since the relevant question for transfer is whether learned ranking outperforms unlearned pruning under the same masking constraint, not whether DQRS is faster than the released implementation that does not prune at all. That comparison is taken up in Section 4.3.2 and revisited in Section 4.4.

4.3 Ablation Study

Two questions remain after the main results. First, how does the choice of top-k trade cost reduction against rewrite overhead? Second, how much of the observed gain comes from pruning itself, and how much from learned ranking? Section 4.3.1 examines top-k. Section 4.3.2 compares DQN-guided ranking against random ranking under the same legal-action mask.

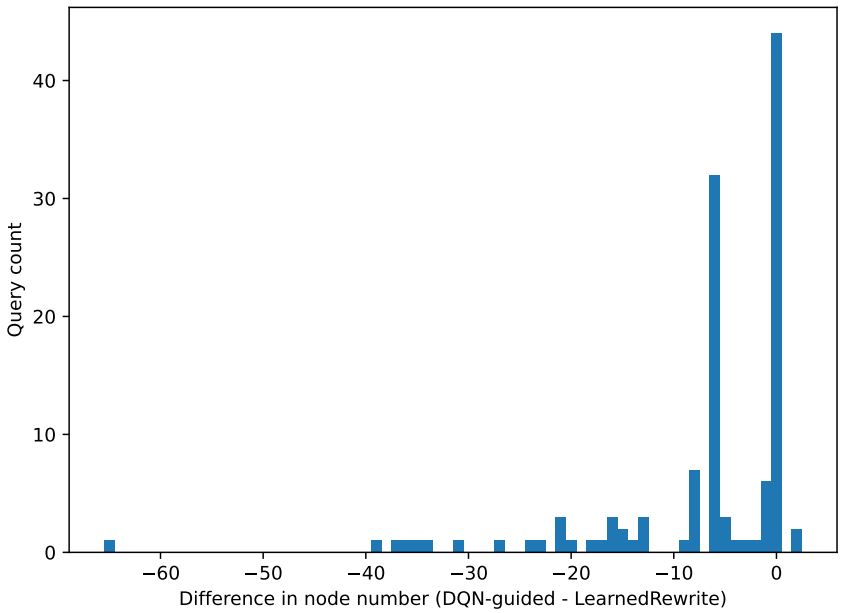

Figure 4.4: Distribution of per-query node-number differences on the TPC-DS-derived workload.

4.3.1 Effect of top-k

Table 4.3 shows the effect of varying top-k on the TPC-H-derived test set. Increasing k from 1 to 2 leaves the average cost reduction unchanged at the reported precision, 4230 in both cases, while the median rises from 103 to 156. This improvement comes at a substantial cost in overhead. Average latency rises from 4.0 ms to 28.9 ms, an increase of roughly a factor of seven; median latency rises from 3.5 ms to 17.1 ms; p90 rises from 5.3 ms to 62.7 ms; and average node number rises from 2.4 to 8.8. On this workload, $k = 1$ is the more favorable DQRS operating point: it preserves average cost reduction close to the $k = 2$ setting while reducing rewrite-search overhead far more aggressively. For this reason, $k = 1$ is used as the main TPC-H setting throughout the chapter. A direct comparison with the released baseline in Table 4.1 sharpens this point. At $k = 2$, the average and median cost reductions of DQRS (4230 and 156) are essentially identical to those of the released baseline (4230 and 156), but the average rewrite latency rises from 15.3 ms (baseline) to 28.9 ms (DQRS at $k = 2$), an increase of approximately 88%. DQRS at $k = 2$ is therefore strictly worse than the released baseline on TPC-H along the latency dimension, while not being measurably better along the quality dimension. The empirical operating range over which DQRS produces a net improvement over the released baseline on this workload is consequently narrow and concentrated at $k = 1$, where the latency reduction is purchased at the cost of a measurable median-quality regression. Larger k values were not explored on TPC-H because the transition from $k = 1$ to $k = 2$ already shows overhead growing well above the baseline level with no compensating quality gain, indicating that further increases in k would not improve the operating point.

Table 4.4 reports the effect of top-k on the TPC-DS-derived validation workload. The trade-off here is more nuanced. Smaller values such as $k = 2$ and $k = 3$ reduce overhead

Table 4.3: Effect of top-k on the TPC-H-derived test set. Bold values indicate the best result for each metric. Values that coincide at the reported precision are not bolded.

Setting	Cost Reduction ↑		Rewrite Latency (ms) ↓			Node Number ↓	
	Avg	Median	Avg	Median	P90	Avg	Median
DQRS, $k = 1$	4230	103	**4.0**	**3.5**	**5.3**	**2.4**	**2.0**
DQRS, $k = 2$	4230	**156**	28.9	17.1	62.7	8.8	4.0

more aggressively but also reduce cost reduction, especially in the median: median cost reduction stays around 713 for both, well below the 1200 achieved by $k = 4$. Average cost reduction is also lower at smaller k. Increasing k to 4 brings cost reduction back to a level close to the released baseline. Increasing further to $k = 5$ does not improve quality further but raises both average latency and average node number. For this reason, $k = 4$ is used as the main TPC-DS setting: it is the smallest value that restores the baseline-level cost reduction without paying the additional overhead of $k = 5$.

Table 4.4: Effect of top-k on the TPC-DS-derived workload. Bold values indicate the best result for each metric; ties are bolded. Values shown as equal may coincide only at the reported precision, and the text distinguishes these cases.

Setting	Cost Reduction ↑		Rewrite Latency (ms) ↓			Node Number ↓	
	Avg	Median	Avg	Median	P90	Avg	Median
DQRS, $k = 2$	96400	713	**366**	**83.6**	**974**	**3.1**	**2.0**
DQRS, $k = 3$	96400	713	2100	715	5630	14.0	9.0
DQRS, $k = 4$	**96600**	**1200**	3360	1110	9000	17.1	15.0
DQRS, $k = 5$	**96600**	**1200**	4990	2210	14000	21.4	15.0

A more granular reading of Table 4.4 reveals a structural property of the TPC-DS-derived workload that is independent of any comparison against the released baseline. The cost reduction achieved by DQRS at $k = 5$ is identical to that at $k = 4$, both on average (96600) and on the median (1200). The two settings are not merely close: the underlying unrounded measurements agree exactly, so the identity is not an artifact of the rounding convention adopted in Section 4.1.4. The equality shown between $k = 2$ and $k = 3$ in Table 4.4, by contrast, is a consequence of the convention: those two settings differ in the underlying measurements, but by less than the reported precision. At the same time, $k = 5$ generates more policy-tree nodes per query on average than $k = 4$ (21.4 vs 17.1, an increase of approximately 25%), while the median node count is unchanged (15 in both). The increase on average node count without a matching increase in median indicates that the additional exploration enabled by $k = 5$ is concentrated on a subset of queries, those whose search trajectories reach states with five or more legal rule classes. The fact that this additional exploration produces no measurable improvement in cost reduction, neither on average nor on the median, places a quantitative ceiling on the cost reduction that

any top-k method can achieve on this workload at $k \geq 4$. Two readings, which are not mutually exclusive, are consistent with this saturation. First, on most expanded states the number of legal rule classes is at most four, so $k = 5$ retains the same children as $k = 4$ on those states, and the additional rule classes retained at the minority of high-legal-action states are systematically not the most cost-reducing ones. Second, the rule-class value distribution under TPC-DS may be such that the rule classes ranked fifth or below by DQRS are rarely the rewrites that close the cost-reduction gap to the optimal rewrite, even when they are legal. Either reading suggests that, under the evaluated setup, top-4 selection on TPC-DS is already close to a workload-level cost-reduction plateau that top-5 selection does not exceed.

The two ablation tables together suggest that the appropriate top-k depends on workload characteristics. On the in-domain TPC-H-derived workload, the value model can identify the relevant rule class confidently enough that $k = 1$ preserves the workload-level average cost reduction. On the cross-workload TPC-DS-derived setting, the model's preferences transfer only partially, and a larger k is needed to compensate by retaining a wider candidate set. This pattern is consistent with the difference in training data exposure: the model is trained only on TPC-H-derived transitions and is evaluated on TPC-DS without further adaptation.

4.3.2 DQN-guided top-k vs random-k

The previous ablation isolates the effect of the top-k value, but it does not separate the contribution of pruning itself from the contribution of learned ranking within the pruned subset. To examine this, DQN-guided ranking is compared against a random-k baseline that selects k rule classes uniformly at random from the legal-action mask, holding everything else fixed. Both methods operate under the same masking constraint, the same rewrite framework, and the same workloads. Differences between them therefore isolate the contribution of the learned ranker.

Table 4.5: DQN-guided top-k versus random top-k under the same masking constraint. Bold values indicate the best result for each primary metric. Values that coincide at the reported precision, and differences that Section 4.5.5 treats as unresolvable, are not bolded.

Workload	**Method**	**Cost Reduction ↑**		**Rewrite Latency (ms) ↓**			**Node Number ↓**	
		Avg	**Median**	**Avg**	**Median**	**P90**	**Avg**	**Median**
TPC-H ($k = 1$)	DQN-guided	**4230**	**103**	4.0	3.5	5.3	2.4	2.0
	random-k	3940	13	**2.0**	**1.7**	**2.8**	2.4	2.0
TPC-DS ($k = 4$)	DQN-guided	96600	1200	**3360**	**1110**	**9000**	17.1	15.0
	random-k	96800	1200	3550	1240	11000	**14.4**	**11.0**

On the TPC-H-derived test set with $k = 1$, the random baseline achieves lower latency than DQRS (2.0 ms versus 4.0 ms on average), but it loses substantially on rewrite quality. Average cost reduction drops from 4230 under DQRS to 3940 under random selection, and

the median collapses from 103 to 13. Node numbers are essentially identical (averages 2.4 for both methods, medians both 2), so the difference is not in how much pruning is done but in which rule class is kept. On this workload, the learned ranker preserves the median cost reduction that random pruning collapses by an order of magnitude, even though the latency overhead of running the model itself is non-zero.

On the TPC-DS-derived workload with $k = 4$, the comparison does not support the value of learned ranking. Random-k achieves a slightly higher average cost reduction (96800 versus 96600 for DQRS, identical to the released baseline value reported in Table 4.2), an identical median cost reduction (1200 for both methods), and fewer policy-tree nodes on average (14.4 versus 17.1, medians 11 versus 15). The only metric on which DQRS is ahead is rewrite latency (averages 3360 ms versus 3550 ms). Within the resolution of this experiment, random selection under the same legal-action mask matches or outperforms DQN-guided ranking on quality and search effort, with DQRS retaining only a modest latency advantage.

On TPC-DS at $k = 4$, the per-node latency relationship reverses: DQRS averages approximately 197 ms/node (3360 / 17.1), while random-k averages approximately 247 ms/node (3550 / 14.4). Two non-mutually-exclusive mechanisms are consistent with this reversal. First, the rule classes preferred by the learned ranker may produce rewritten states whose subsequent EXPLAIN evaluation is systematically cheaper than those reached under random selection. Second, random selection may more frequently invoke rule classes that fail acceptance checks after rewrite execution but before child insertion, contributing latency without contributing nodes. This holds despite DQRS carrying the additional per-expansion EXPLAIN call described in Section 3.2.1, which random-k does not perform. Distinguishing these mechanisms would require per-stage timing breakdown that is not available in the present experiments.

Two independent identities in the TPC-DS cost-reduction results jointly characterize the regime in which this comparison is conducted. Both identities are exact in the underlying measurements and are not consequences of the reporting precision adopted in Section 4.1.4. The first, observable directly from Table 4.4 and discussed in Section 4.3.1, is that DQRS at $k = 5$ achieves identical cost reduction to DQRS at $k = 4$ despite expanding approximately 25% more policy-tree nodes per query on average. The second, observable from Tables 4.2 and 4.5, is that random-k at $k = 4$ achieves an average cost reduction of 96800, which coincides exactly with the value obtained by the released baseline (Table 4.2) in the underlying unrounded measurements. Taken together, these two identities suggest that the evaluated $k = 4$ setting on the TPC-DS-derived workload is close to a workload-level cost-reduction saturation regime, in which every top-4 method evaluated approaches a similar ceiling. Small differences between methods near this ceiling therefore carry less interpretive weight than differences observed away from saturation.

Within this saturation context, two readings of the random-k finding remain consistent with the data. The first is that the learned ranker fails to transfer to the TPC-DS-derived workload, since the model is trained only on TPC-H-derived transitions (Section 4.1.5) and the rule-class value structure may differ between the two workloads; this would explain why DQN-guided selection at $k = 4$ reaches a slightly lower cost reduction than random-k at the

same k (96600 vs 96800, a relative difference of approximately 0.1%). The second is that, since the comparison is occurring near a workload-level ceiling, the small directional gap between the two methods does not, by itself, support a strong transfer-failure conclusion. The two readings are not separable by the present experiments. Distinguishing them would require evaluating both methods at a k value below the saturation point on TPC-DS, such as $k = 2$ or $k = 3$. Table 4.4 shows that DQRS produces visibly lower cost reduction at these settings than at $k = 4$. The random-k comparison would then need to be replicated at the same operating point. Such an evaluation is left to future work.

Taken with the TPC-H result, the random-k comparisons support the value of learned ranking only in the in-domain (TPC-H) setting. The TPC-DS comparison is therefore reported as inconclusive with respect to cross-workload transfer, with the additional caveat that the comparison at $k = 4$ may not be structurally informative on this workload. It should not be interpreted as a robustness confirmation.

4.4 Analysis

The pattern across the experiments is consistent. DQRS does not improve final cost reduction beyond the released implementation; the goal it pursues is to reduce rewrite-search overhead while keeping cost reduction close to that implementation, and the experiments confirm this in aggregate. On the in-domain TPC-H-derived test set, average cost reduction is essentially unchanged while latency and node number drop substantially. The median cost reduction drops on the same workload, so the aggregate-level statement should be read together with the typical-case statement: under top-$k = 1$ on TPC-H, DQRS preserves the workload-level average and gives up roughly a third of the median cost reduction in exchange for the efficiency gains. On the cross-workload TPC-DS-derived setting, the same direction holds, with cost reduction unchanged on average and a clearly visible but smaller reduction in overhead. The thesis claim is therefore that DQRS is a search-efficiency mechanism within the existing framework, not a quality-improving rewriter.

The contrast between the two workloads should be read together with the random-selection comparison in Section 4.3.2. On the in-domain TPC-H-derived test set, the value model prunes aggressively (top-$k = 1$) while preserving the average cost reduction of the released implementation, at the cost of an approximately 34% reduction in median cost reduction. Under the same $k = 1$ setting, random selection collapses median cost reduction by an order of magnitude (Table 4.5), so the learned ranker is doing work at this operating point that is not reducible to pruning alone. The contribution of the learned ranker is therefore best characterized as median cost-reduction preservation under aggressive top-1 pruning, rather than as an across-the-board efficiency improvement: at $k = 2$, where the median cost reduction recovers to the baseline level, the latency advantage of DQRS over the released baseline is no longer present (Section 4.3.1). On TPC-H, DQRS does not Pareto-dominate the released baseline; it occupies a different point on an efficiency–quality trade-off curve.

A more granular reading of the $k = 2$ row in Table 4.3 reveals a pattern that is not captured by the design rationale in Section 3.3.4. DQRS at $k = 2$ generates marginally fewer

policy-tree nodes than the released baseline on average (8.8 vs 9.9) and an identical number in median (4 vs 4), yet its average rewrite latency is roughly 1.9× that of the baseline (28.9 ms vs 15.3 ms). The per-node latency therefore approximately doubles between the two methods: DQRS at $k = 2$ spends roughly 3.3 ms per generated node, against roughly 1.5 ms for the released baseline. Two non-mutually-exclusive mechanisms are consistent with this observation. First, the per-state encoding, legal-action mask construction, and ONNX Runtime forward pass collectively add a per-node overhead. One component of this overhead is identifiable rather than inferred: constructing the state encoding requires the optimizer's cost estimate for the current state, which the implementation obtains through an EXPLAIN call issued once per expansion and which neither the released baseline nor random-k performs. Its magnitude is not measured separately, but it is a database round-trip rather than in-process computation, so it is unlikely to be negligible at small k. Second, by selecting different rule classes than the baseline at the same node, DQRS may steer the search into rewritten states whose subsequent rewrite execution and EXPLAIN evaluation are individually more expensive, even when the total node count is no larger. Quantifying the relative size of these two contributions would still require a per-node timing breakdown of encoding, inference, masking, and EXPLAIN that is not available in the present experiments. The observation is reported here as a quantitative constraint on where DQRS is competitive against the released baseline rather than as a resolved attribution: on TPC-H, the inference and search-shape overheads are amortized by aggressive pruning at $k = 1$ and are not amortized at $k = 2$.

On the cross-workload TPC-DS-derived setting, the same in-domain evidence for the value of learned ranking is absent, but the absence is best interpreted in light of a structural feature of this workload made visible in Table 4.4: DQRS at $k = 5$ achieves the same cost reduction as DQRS at $k = 4$ despite expanding approximately 25% more policy-tree nodes on average. Top-4 selection on this workload therefore already approaches a workload-determined cost-reduction ceiling that further exploration does not exceed. Combined with the identity between random-k at $k = 4$ and the released baseline noted in Section 4.3.2, this implies that the cross-workload random-k comparison reported in this thesis is conducted in a saturation regime: every top-4 method tested approaches a similar ceiling on cost reduction, leaving little headroom in which a learned ranker could distinguish itself from random selection on cost reduction. The DQN-guided method's slightly lower cost reduction at $k = 4$ (96600 vs the random-k and baseline value of 96800, a difference of approximately 0.1%) is a small directional signal that the learned preferences may mildly mis-rank under the TPC-DS distribution, but the magnitude is small relative to the ceiling and does not, on its own, establish a transfer failure. Conversely, the present experiments do not establish that the learned ranker successfully transfers either. The TPC-DS results should therefore be reported as inconclusive on transfer rather than as evidence either for or against it. The reduction in rewrite latency that DQRS does achieve on TPC-DS at $k = 4$ (24% relative to the released baseline, Table 4.2) is consistent with model-induced visit-order changes producing different state distributions during search, but its origin is not separately attributed in the present experiments.

The top-k ablation clarifies how aggressively pruning can be applied. On the in-domain workload, moving from $k = 1$ to $k = 2$ produces only a small improvement in cost reduction

but a large jump in overhead, which is what makes $k = 1$ favorable on TPC-H. On the cross-workload setting, smaller k values reduce overhead more aggressively but degrade quality, especially in the median; only at $k = 4$ does cost reduction return to the baseline level. The appropriate pruning aggressiveness is therefore workload-dependent: in-domain settings can tolerate aggressive pruning, while workload-shifted settings require a less aggressive setting to compensate for partial transfer.

The random-k comparison adds a more nuanced view of the learned ranker itself. On the TPC-H-derived workload, random pruning achieves lower latency than DQRS but causes a sharp drop in cost reduction, especially in the median. This is the cleanest evidence that the learned ranker is doing something that random selection cannot replicate: it preserves the median-level cost reduction that random pruning collapses. On the TPC-DS-derived workload, the random-k comparison is inconclusive with respect to the value of learned ranking. The learned ranker has a better latency profile, but the random baseline reaches slightly higher cost reduction and uses fewer nodes in this run. As discussed in Section 4.3.2 and earlier in this section, this comparison occurs near a workload-level cost-reduction saturation regime, which structurally limits its discriminating power independently of seed variability. The main support for the value of learned ranking therefore comes from the TPC-H-derived case, where the comparison is conducted away from any visible saturation. A more granular reading of Table 4.5 distinguishes which dimension of cost reduction the learned ranker preserves on TPC-H. The average cost reduction gap between DQN-guided ranking and random selection at $k = 1$ is moderate: 4230 versus 3940, a relative difference of approximately 6.8%. The median, in contrast, separates the two methods by nearly an order of magnitude: 103 versus 13. Because both methods produce essentially the same number of policy-tree nodes (averages 2.4 vs 2.4), the difference is not in how aggressively pruning is applied but in which rule class is retained when multiple are legal. The contribution of the learned ranker on TPC-H is therefore concentrated in the median rather than spread across the workload.

The latency reductions and node-number reductions track each other consistently across both workloads. Latency reductions are slightly smaller than node-number reductions in absolute percentage on TPC-H, about 74% against about 76%, but in the same direction, while on TPC-DS the two reductions are similar in scale. This pattern is what one would expect from the design of DQRS: pruning rule classes during expansion reduces the number of children generated, and each saved child saves a corresponding cycle of rule application, executability checking, and EXPLAIN cost evaluation. Node number therefore provides a structural proxy for part of the observed latency reduction, although it does not fully determine latency because per-node costs can differ across states and methods. Within this study, the value of learned ranking is supported on the in-domain TPC-H-derived workload and remains inconclusive on the TPC-DS-derived workload. Pruning aggressiveness should be chosen with respect to the workload, and the appropriate setting under workload shift remains an open question.

4.5 Scope and Validity Considerations

This section summarizes the main scope and validity considerations for the empirical study. The discussion is organized into four categories: workload construction and tuning protocol, baseline scope and quality signal, training data and offline reinforcement learning, and evaluation reliability and implementation scope. Each category identifies how the corresponding design choices affect the interpretation and generality of the reported results, and points to the mitigation or extension that would be needed in future work.

4.5.1 Workload construction and tuning protocol

Workload filtering and representativeness. Both workloads are filtered subsets, not standard benchmark streams: the TPC-H-derived workload is a 668-query subset of the LearnedRewrite-released synthetic pool (split 468/200 for training and held-out test), and the TPC-DS-derived workload is a corrected and filtered 123-query subset. Both retain only queries with non-zero baseline cost reduction, which biases typical-case quality comparisons against any method that lowers per-query quality: the retained queries are precisely those on which the released baseline already rewrites well. The 34% median drop reported in Section 4.2.1 should be read in that light. Conversely, queries on which the released baseline produces zero cost reduction were excluded by this filter, so any case in which DQRS would produce a non-zero rewrite where the baseline does not is also outside the scope of the reported comparison. The filter therefore removes both directions of asymmetric performance from the evaluation, not only cases unfavorable to DQRS. Reported results apply within these filtered settings, not to arbitrary official benchmarks or production workloads.

Top-k tuning protocol. The top-$k = 1$ (TPC-H) and top-$k = 4$ (TPC-DS) settings used in the main results are selected from the same workloads on which Section 4.2 reports those results, rather than from a separately tuned validation split. The reported numbers are therefore best read as operating-point characterizations at empirically chosen k values, not as estimates of generalization performance under independently tuned operating points. Tuning k on a held-out validation split disjoint from main-result evaluation is identified as future work.

4.5.2 Baseline scope and quality signal

EXPLAIN cost as quality proxy. Rewrite quality is measured by PostgreSQL EXPLAIN cost, the same estimator the released baseline uses internally; this keeps the within-framework comparison consistent. EXPLAIN cost is, however, an optimizer-side proxy rather than a direct measurement of execution time. This choice is implementation-aligned: both the released baseline and DQRS use PostgreSQL EXPLAIN cost internally for candidate evaluation and acceptance, while the intervention studied here targets rewrite-search overhead rather than physical query execution. Reported quality results should therefore

be read as estimator-side improvements under the current setup, not as guarantees of end-to-end runtime improvement on all deployments. Measuring true execution time would require a separate controlled execution protocol, including query execution, repeated runs, timeout handling, and cache or warm-up control; this evaluation is identified as future work.

Released baseline vs the system proposed in the paper. The baseline used throughout this thesis is the released LearnedRewrite single-rule implementation, which does not include the learned cost estimator described in the paper (Section 3.1.3). Reported gains are therefore relative to a structurally faithful but learning-guidance-stripped instantiation, not to the full system proposed in the paper. Whether the same magnitude of improvement would hold against a full reproduction of the system proposed in the paper, and how DQRS would interact with the paper's estimator if both were active, remain open and are flagged as future work.

4.5.3 Training data and offline reinforcement learning

In-domain vs out-of-domain training distribution. The DQN model is trained only on transitions collected from the 468-query TPC-H-derived training split. The TPC-H-derived test set is in-domain; the TPC-DS-derived workload is out-of-domain and tests cross-workload behavior. The TPC-DS results should therefore be read as transfer-style validation, not as the best achievable performance under a TPC-DS-trained model. A model trained directly on TPC-DS-derived transitions would likely perform differently and is an obvious next step.

Behavior-policy distribution shift. Training transitions are collected by running the released search procedure, which expands every legal rule class at each selected node (Section 3.3.1). At inference, DQRS expands only the top-k ranked rule classes per node, so the search visits a different distribution of policy-tree states than the model saw during training. This is a known offline-RL concern: a value model accurate on training-distribution states is not guaranteed to remain accurate under a different action-selection policy. The thesis does not apply importance reweighting, behavior-policy correction, or conservative value estimation to mitigate this. The inconclusive cross-workload comparison in Section 4.3.2 is consistent with such a shift but is not isolated from workload-level distributional difference or from the top-k saturation regime discussed in Section 4.5.4.

Accepted-transition sampling bias. The dataset records only candidate children that are successfully generated and accepted by the released rewrite routine. Legal rule-class attempts that fail to produce a valid child or are rejected by existing filtering are not represented as explicit negative transitions. The model therefore observes accepted-action outcomes more directly than failed or filtered-action outcomes, which can bias the learned value function toward rule classes that appear in accepted transitions and leaves some unproductive legal actions under-penalized. The legal-action mask prevents illegal actions at inference but does not by itself teach the model that a legal but unproductive action should be down-ranked when that action is absent from the dataset. A specific consequence is that the model never observes a transition in which a legal rule class produces no

accepted child; all training samples are positive in the sense that an accepted child exists. The model therefore has no direct supervision against ranking unproductive-but-legal rule classes highly when they are masked-in at inference, beyond what the positive transitions implicitly disprefer through their relative reward magnitudes.

Terminal-flag labeling trade-off. As described in Section 3.3.1, the terminal flag is not assigned when the search budget is exhausted. Budget exhaustion is treated as search truncation rather than rewrite terminality. Instead, a transition is marked `done` = `true` only when its next state is actually selected for expansion and produces no accepted child under the implemented rewrite and filtering checks. This design avoids falsely suppressing the bootstrap term for states whose continuation value is merely unobserved because of UCT selection or the finite search budget. The data-collection wrapper additionally terminates the search early when all current leaf nodes have been selected without producing a new accepted child, so the convention is implemented consistently rather than approximated. The remaining limitation of this convention is conservative false non-terminal labeling. A transition whose next state is never selected for expansion retains `done` = `false`, even if that next state would have produced no accepted child had it been tested. Consequently, the bootstrap term in Equation 3.4 contributes to the training target on some transitions where it would be suppressed under complete terminal-state observation, biasing learned values toward optimistic attribution at under-tested states. Within the action-ranking role DQRS plays, this is acceptable as long as the rule-class ordering at the current state remains well-calibrated against per-state rewards: ranking does not require unbiased absolute Q-values, only consistent relative ordering. The same bias would be a more serious concern in a setting that depended on unbiased absolute value estimates, such as combining DQRS scores with a reproduced selection-stage estimator.

Train-inference legal-mask asymmetry. At inference, the legal-action mask suppresses illegal actions before top-k ranking. At training, the recorded transition format stores the encoded next state but not its legal-action mask, so the maximum in Equation 3.4 is taken over all seven actions. The bootstrap target can therefore include action-value estimates that would be suppressed at deployment, and is at least as large as a mask-restricted target would be (taking max over a superset cannot decrease the maximum). The empirical magnitude is not directly measured here; mitigation would require re-collecting transitions with the next state's mask included, or recomputing masks offline from the recorded relational state. Both are identified as future work.

4.5.4 Evaluation reliability and implementation scope

Stochasticity, single-seed random-k, and TPC-DS saturation. Cost reduction and node number are deterministic given a fixed search procedure and database state and are evaluated once per query. Rewrite latency is averaged over three runs per query, which reduces but does not eliminate system noise. The random-k baseline introduces a separate variability source: it selects k rule classes uniformly at random from the legal-action mask, and reported random-k results reflect a single controlled run, not a distribution over seeds. The TPC-H random-k comparison is less likely to be overturned by seed variation because

the median cost-reduction gap is large, but this is not directly verified. The TPC-DS comparison is more sensitive to seed variability because the methods are much closer; in addition, as analyzed in Sections 4.3.1, 4.3.2, and 4.4, the $k = 4$ comparison on TPC-DS occurs in a saturation regime in which DQRS at $k = 5$ matches DQRS at $k = 4$ and random-k at $k = 4$ matches the released baseline. The discriminating power of the TPC-DS comparison is therefore structurally limited, independently of seed variability. A discriminating cross-workload evaluation would require evaluation at a k value below the saturation point and across multiple seeds.

Implementation scope. The study is conducted on the released LearnedRewrite single-rule branch under a fixed rewrite budget of 20. These settings are appropriate for a controlled module-level comparison but constrain the range of conclusions. Results support the value of DQN-guided rule-class selection within the in-domain TPC-H-derived setting under this implementation context. They do not support stronger claims about other LearnedRewrite variants, other rewrite budgets, other deployment environments, or workload distributions beyond the in-domain setting evaluated here.

Action-space granularity. DQRS operates over the seven rule-class actions exposed by the released candidate-generation interface, as motivated in Section 3.2.2. Two alternative granularities were considered but not evaluated empirically: a finer per-instance action space over the 57 concrete Calcite rule instances grouped under the seven classes, and a coarser binary node-level gate. The choice of the rule-class granularity is an alignment choice with the released interface and with the scope of the offline transition data, not an empirically validated optimum. Whether a per-instance action space would yield finer ranking signal at the cost of larger output dimension and sparser supervision per action is an open empirical question. An empirical comparison would require modifying the released candidate-generation routine to expose per-instance hooks, re-collecting transitions at the per-instance level, and re-training the value model; this is identified as future work in Section 5.2. Within the present work, claims about the value of DQN-guided rule-class selection are therefore conditional on the rule-class-level action space and do not extend to the question of which granularity is empirically optimal.

4.5.5 Reporting precision and the absence of significance testing

The comparisons reported in this chapter are descriptive. No hypothesis testing is performed and no confidence intervals are estimated, for two reasons: the two deterministic methods produce a single value per query rather than a sampling distribution, and the cost-reduction distributions are strongly heavy-tailed, with a small number of high-impact rewrites dominating every workload-level mean. Under these conditions the reported averages are not accompanied by a resolution at which small differences could be declared meaningful.

This constrains which of the reported differences can bear interpretive weight. Differences of a fraction of a percent in average cost reduction, 4227 versus 4231 on the TPC-H-derived workload and 96639 versus 96758 on the TPC-DS-derived workload in the unrounded measurements, are smaller than the corresponding standard deviations by more

than three orders of magnitude. On the TPC-H-derived workload the two values coincide at the precision reported in Table 4.1. On the TPC-DS-derived workload they remain distinguishable at the third significant figure of Table 4.2, but the difference is likewise far below the resolution the measurements support. Both are treated throughout as indistinguishable rather than as evidence of an advantage in either direction. The claims defended in this thesis rest instead on differences whose magnitude is large relative to the dispersion of the metric: the reductions in rewrite latency and policy-tree size on both workloads, the near-order-of-magnitude gap in median cost reduction between learned and random selection at $k = 1$ on TPC-H (103 versus 13), and the exact saturation identities on TPC-DS discussed in Section 4.3.2. Statements elsewhere in this chapter that one method preserves or matches the cost reduction of another should be read in this sense as the absence of a resolvable difference and not as a positive claim of equivalence.

4.6 Learned Ranking vs Hand-Crafted Heuristics

The experiments in this chapter compare DQRS against the released implementation and against random selection under the same legal-action mask. Neither comparison addresses a question that is natural to ask of any learned component in a query rewriter: does the learned ranking encode anything that a carefully designed hand-crafted rule ordering could not? No hand-crafted heuristic baseline was implemented in this work, so this section does not report an experimental comparison. It is a qualitative discussion that contrasts the criterion a fixed rule ordering is able to express with the criterion the trained model appears to apply, using the statistics of the recorded training transitions together with the model's own preference distribution. The observations below are interpretations of measurements already reported rather than results of a separate experiment.

As described in Section 2.2.1, a traditional rule-based rewriter decides rule order by a fixed traversal or a predefined sequence, and a rule is attempted when its pattern matches the current plan fragment. Two properties of this arrangement matter here. First, the ordering is decided once, at design time, and is therefore the same at every state the rewriter encounters; it cannot condition on the particular query being rewritten. Second, and more fundamentally, pattern matching establishes only that a rule is applicable. The designer may place empirically useful rules earlier in the sequence, and may also encode known interactions between rules. What the ordering cannot do is change with the query: the same relative priority applies at every state the rewriter encounters.

Two of the three rule classes that appear in the training data are ranked by the model in a way a well-informed hand-crafted ordering would reproduce. Table 4.6 summarizes three quantities computed over the 4,169 transitions recorded from the 468 TPC-H-derived training queries described in Section 4.1.6. The first column gives each class's share of those transitions. The second gives the proportion of the stored state vectors at which the trained network assigns that class the highest Q-value among the three classes that occur in the training data. The third gives the proportion of attempts in which applying the class lowered the estimated cost. The remaining four rule classes do not appear in the training data at all. The three classes are used at comparable rates by the exhaustive

baseline, at 38.7%, 32.3% and 29.0% of recorded transitions, so the frequency with which a class is attempted carries little information about its usefulness. The model's preference distribution is markedly less even. Filter rules, which lowered the estimated cost in 42.3% of the attempts in which they were applied, are ranked first substantially more often than their share of the data. Project rules, which lowered the cost in only 2.1% of attempts, are ranked first far less often than their share of the data. The model has therefore not reproduced the distribution of the policy that generated its training data; it has moved weight toward the class with a positive empirical signal and away from the class without one. A hand-crafted ordering informed by the same evidence would order these two classes the same way. On this dimension the learned ranking does not contribute anything a designer could not have supplied.

Table 4.6: Rule-class usage, learned preference, and empirical cost-reduction rate on the training transitions.

Rule Class	Share of Transitions	Ranked First	Lowered Cost
rule_filter	38.7%	62.1%	42.3%
rule_join	32.3%	31.6%	0.0%
rule_project	29.0%	6.2%	2.1%

The third class separates the two approaches. Join rules never produced a positive immediate reward in this workload: their mean reward is negative, and none of the 1,347 recorded attempts lowered the estimated cost of the state they were applied to. Judged on immediate benefit alone, join rules would be ordered last. The trained model nonetheless ranks them first in roughly a third of the states in the training set, a proportion close to their share of the data and far above what their immediate reward would justify. The training objective accounts for this. The quantity the network estimates is not the reward of a rewrite step but the value of the state that step leads to, defined by the bootstrap term in Equation 3.4: an action whose own reward is zero can still carry value if the states it makes reachable have high value. A join rewrite that does not itself reduce the estimated cost may restructure the plan so that a subsequent filter rewrite becomes applicable, and the value of that subsequent step is propagated back through training. This is where the two approaches differ. A designer can encode known rule interactions in a fixed ordering: if a class is understood to enable a later one, it can be placed earlier in the sequence, and hand-crafted orderings in production systems do exactly this. What a fixed ordering cannot do is vary that priority with the query in front of it. It assigns the same relative order at every state, whereas the ranking learned here is conditioned on features of the current state and is derived from recorded outcomes rather than from a designer's prior understanding of the rule set. The difference is therefore one of state dependence and of where the ordering knowledge comes from, not one of expressive power.

Three consequences follow for the integration of learned components into the rewriting mechanisms of a database system. First, the natural point of integration is an ordering layer above an existing rule engine rather than a replacement for the rule set. DQRS is an instance of this arrangement: the rules themselves, the executability check and the

cost-acceptance condition are unmodified, and the learned component decides only which candidates are attempted. This bounds, but does not eliminate, the consequence of a mis-ranking. A wrong ranking cannot introduce a rewrite that the existing acceptance conditions would have rejected, so correctness is unaffected. It can, however, discard a branch that would have led to a better rewrite, and the drop in median cost reduction reported in Section 4.2.1 is an instance of exactly that cost. For a production optimizer, where correctness and predictability constrain what can be replaced, containing the failure mode to quality rather than correctness is more important than the accuracy of the model itself. Second, the value of a learned ordering is concentrated rather than uniform. It arises where the useful ordering depends on the query rather than being uniform across the workload. For the filter and project classes in Table 4.6, the learned preference points in the same direction that their recorded outcomes would suggest, so a state-independent ordering informed by the same statistics would plausibly rank them the same way; no hand-crafted baseline was implemented, so this is an expectation rather than a measured comparison. This suggests a criterion for deciding whether a learned component is worth introducing to a given rewriting mechanism: the expected gain depends on how much of the rewrite benefit in the target workload is realized through sequences of rules rather than through individual rules. Third, whether the integration pays for itself is a question of cost ratio rather than of model quality. The learned component adds a per-state encoding, mask construction and inference cost at every expansion, and it must displace more search work than it introduces. In the in-domain setting of this study the end-to-end measurement settles the question in aggregate: average rewrite latency falls from 15.3 ms to 4.0 ms while average policy-tree size falls from 9.9 to 2.4 accepted nodes, so the total overhead introduced by the selection mechanism is more than offset by the search work it displaces. The relative size of its components is not separately measured; Section 4.4 notes that a per-node breakdown of encoding, inference, masking and EXPLAIN is not available in the present experiments. That balance is not a property of the model. As Section 4.4 shows, the same inference cost is no longer amortized at $k = 2$, where the node-count reduction is small. The condition for integration is therefore that the cost of one inference be small relative to the cost of the node it avoids, and that condition must be checked for the target system rather than assumed.

These readings are interpretations of measurements reported elsewhere in this chapter and are subject to the corresponding limitations. They rest on a single workload and a single framework, and the behaviour of join rules may reflect the structure of TPC-H-derived queries rather than a general property of the class. Because no hand-crafted heuristic was implemented, the ordering such a heuristic would produce is inferred from the criterion available to it rather than observed. The model's preference distribution is measured on the states recorded during training rather than on held-out queries. Finally, attributing the ranking of join rules to the bootstrap term is consistent with the training objective but is not established by a controlled comparison; separating that contribution would require the gamma-zero ablation described in Section 3.3.2, which was not performed.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis addresses the efficiency of learning-guided query rewrite under a fixed rule set. The starting point is an observation about the released LearnedRewrite single-rule implementation: expansion at each search node is performed by enumerating all applicable rule classes, and each candidate triggers concrete rewrite execution and cost evaluation. The cost of generating candidates therefore contributes directly to overall rewrite latency, rather than being a free branching step. From this observation, the thesis isolates expansion-time rule-class selection as a distinct intervention point and frames it as a state-dependent decision problem: at each search state, the system chooses which rule classes to expand from a finite, state-dependent set of legal alternatives.

To address this problem, the thesis proposes DQRS, a DQN-guided top-k selective expansion method. The method preserves the surrounding rewrite framework and replaces only the exhaustive enumeration of legal rule classes at expansion. The current rewrite state is encoded as a fixed-length feature vector summarizing structural and coarse cost-related properties of the relational tree. A Q-network trained offline over rewrite transitions estimates action values for the rule classes legal at the current state, and only the top-k ranked legal rule classes are forwarded to the existing single-rule rewrite routine for expansion. The model is trained from transitions collected by instrumenting the released search procedure, and the trained model is integrated into the rewrite framework as an inference-only component during online query rewriting.

The empirical evaluation in Chapter 4 indicates that DQRS primarily affects the efficiency–quality trade-off of rewrite search rather than dominating the released implementation along all dimensions. On the held-out TPC-H-derived test set, DQRS at $k = 1$ preserves the workload-level average cost reduction, reduces the median cost reduction by approximately 34%, and reduces average rewrite latency and average policy-tree size by approximately 74% and 76%, respectively. Increasing k to 2 on the same workload restores the median cost reduction to the baseline level but eliminates the latency advantage, indicating that the efficiency improvement of DQRS on TPC-H is operating-point-specific

rather than universal. The value of learned ranking over random selection on this workload is supported by a substantial gap in median cost reduction at $k = 1$ (103 vs 13), which is not reducible to pruning alone. The $k = 1$ (TPC-H) and $k = 4$ (TPC-DS) operating points reported above are selected on the same workloads on which they are evaluated, so the reported figures characterize empirically chosen operating points rather than estimates under independently tuned k values; this scope condition is discussed as part of the validity considerations in Section 4.5.1.

On the TPC-DS-derived validation workload, DQRS at $k = 4$ reduces average rewrite latency by approximately 24% and average policy-tree size by approximately 30% while keeping average and median cost reduction within 0.2% of the released baseline. However, under the same legal-action mask, random selection at $k = 4$ also matches the released baseline on cost reduction, and the analysis in Section 4.3.2 shows that DQRS at $k = 5$ reaches the same cost reduction as DQRS at $k = 4$ despite expanding more nodes. The cross-workload random-k comparison is therefore conducted near a top-k saturation regime in which every evaluated method approaches a common workload-level ceiling. The present experiments do not establish that the learned ranker contributes to rewrite quality on the TPC-DS-derived workload, and cross-workload transfer of the learned preferences remains unresolved.

Two scope conditions shape these findings. The baseline used throughout the evaluation is the released LearnedRewrite single-rule implementation, which does not include the learned cost estimator described in the paper. Reported gains are therefore relative to this released baseline rather than to a full reproduction of the system proposed in the paper. The quality signal used in evaluation is PostgreSQL EXPLAIN cost, the same estimator used by the released implementation itself, so quality results are estimator-side improvements rather than direct measurements of true execution time. Both conditions are discussed as scope and validity considerations in Section 4.5 and revisited as future work directions below.

Taken together, the work supports a scoped claim: within the released LearnedRewrite single-rule codebase, a state-dependent learned intervention at the expansion stage can produce a measurable efficiency gain at a specific operating point on the in-domain workload, while exposing an explicit efficiency–quality trade-off. The cross-workload transfer of the learned preferences is not established by the present experiments. The contribution is narrower than a new framework for learned rewriting and broader than an isolated heuristic; it is a focused module-level improvement to a specific decision point inside an existing rule-constrained learning-guided framework.

5.2 Future Work

A natural first direction is to reproduce the learned cost estimator from the LearnedRewrite paper and re-run the evaluation against the resulting reconstructed baseline, which mirrors the paper's full system. The two interventions, the paper's selection-stage estimator and DQRS's expansion-stage ranker, operate at conceptually different points in the search loop

and are not in competition. A combined system would let the estimator guide which node to expand and DQRS guide which rule classes to try at the chosen node. Whether the efficiency gains reported in this thesis carry over to that combined setting, and whether the two components interact constructively or interfere, are open questions that the present scope cannot answer.

A second direction is to extend the evaluation from estimator-side cost reduction to end-to-end execution time. The current results show that DQRS preserves estimated cost reduction close to the released baseline, but estimated cost is not a direct measurement of runtime. Running the rewritten queries produced by both methods on the target DBMS and measuring true execution time, ideally on multiple hardware setups, would clarify whether the estimator-side conclusions translate to deployable runtime improvements. This direction also opens space for re-examining the per-query trade-off: queries where DQRS lowers estimated cost may behave differently at runtime, and queries where it does not improve estimated cost may still benefit from reduced rewrite latency.

A third direction concerns cross-workload generalization, and is the natural extension of the limitation noted in Sections 4.3.2 and 4.5.3. The current model is trained only on transitions collected from the 468-query TPC-H-derived training split, and the cross-workload comparison on the TPC-DS-derived validation set is conducted at $k = 4$, in a regime where every evaluated top-4 method approaches a similar workload-level cost-reduction ceiling. As noted in Section 4.3.2, replicating the random-k versus DQN-guided comparison at a smaller k, such as $k = 2$ or $k = 3$, would let the contrast escape the saturation regime and produce a structurally informative measurement. A result obtained from such an additional evaluation, however, would still characterize a single TPC-H-trained model on a single TPC-DS-derived workload. A substantive answer to whether the rule-class preferences learned by DQRS transfer across workloads requires changing the training distribution rather than only adding evaluation points on a single-workload-trained model. The natural next step is therefore to train DQRS on transitions drawn from a broader pool covering multiple workloads, or to train jointly across workloads with explicit cross-workload validation on a held-out workload disjoint from the training set. Reward formulations beyond per-step EXPLAIN cost change could be examined in the same context, including rewards that incorporate downstream value signals or runtime feedback.

A fourth direction concerns the methodological characterization of the present setup. A direct measurement of the distribution of legal-action counts per state on each workload would confirm the saturation analysis in Sections 4.3.1 and 4.3.2 and would further distinguish whether saturation reflects a structural absence of legal high-k actions or a value-distribution effect at high-k legal actions. Replacing the transition-level training and validation split described in Section 3.3.2 with a query-grouped split would isolate the validation signal from intra-query correlation in the training subset and make early-stopping model selection a stricter check on generalization.

A fifth direction is to ablate the contribution of the DQN bootstrap term against a pure supervised regression baseline trained on the same per-step EXPLAIN-cost reward and the same state encoding. Under the short-horizon data-collection setup of this thesis,

the two formulations may produce similar effective rankers, but the comparison has not been conducted. Resolving it would clarify which components of the DQN training pipeline contribute to the observed behavior and would inform the design of any extension that incorporates a downstream-value signal.

A sixth direction is to compare the rule-class-level action space adopted in this thesis against finer-grained alternatives. The present action space, with seven actions corresponding to the rule classes exposed by the released candidate-generation interface, is an alignment choice with the released interface and with the scope of the offline transition data, not an empirically validated optimum. Two natural alternatives are a per-instance action space over the 57 concrete Calcite rule instances grouped under the seven classes, exposing each concrete rule as its own action, and an intermediate sub-class grouping, for example splitting `rule_filter` into push-down, simplification, and merge sub-actions or splitting `rule_join` along similar operator-pattern lines. Either alternative would require modifying the released candidate-generation routine to expose per-instance or per-sub-class hooks, re-collecting transitions at the corresponding granularity, and re-training the value model. The resulting empirical comparison would test directly whether the rule-class granularity adopted in this thesis is favorable on the rewrite-search-overhead and rewrite-quality metrics used here, or whether a finer granularity yields a better efficiency–quality trade-off.

A seventh direction is to use a large language model as the source of rule-class preference at expansion time. The ranking decision studied here is not intrinsically tied to a value-based reinforcement learning model: a language model could be prompted with a serialized form of the current relational tree together with the set of legally applicable rule classes, and asked to rank them. Because the legal-action mask, the rewrite routine, and the executability and cost-acceptance checks would remain unchanged, a poor ranking would cost search efficiency rather than equivalence; such a ranker would also require no workload-specific training set, which is the origin of the unresolved cross-workload transfer discussed in Section 4.3.2. The obstacle is granularity. The LLM-driven rule selection surveyed in Section 2.5.3 makes one decision per query, whereas the decision point studied here recurs at every expanded node, and the rewrite searches measured in Chapter 4 complete in 4.0 to 15.3 ms on average on the TPC-H-derived workload (Table 4.1), so a per-node model call would be dominated by its own inference cost. Two framings avoid this: a single pre-search call producing a query-level prior that the search reuses at every node, and offline distillation, in which a language model labels rule-class preferences over the transitions recorded in Section 3.3.1 and a small ranker is trained on those labels, paying the inference cost once at training time.

A final direction is to extend the design of the value model itself. The current state representation is a compact hand-crafted feature vector, and the Q-network is a lightweight multi-layer perceptron. Both choices are appropriate for a controlled first study but are not necessarily the most expressive. Future work could examine richer relational-state representations, more expressive value models, or adaptive top-k policies that vary pruning aggressiveness per state instead of fixing a workload-level setting. Hybrid designs that combine learned rule-class selection with other sources of guidance, such as richer optimizer feedback, would be a further step in the same direction.

References

[1] Qiushi Bai, Sadeem Alsudais, and Chen Li. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. *PVLDB*, 16(11):2911–2924, July 2023.

[2] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proc. of SIGMOD*, page 221–230, New York, NY, USA, 2018. ACM.

[3] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[4] Sriram Dharwada, Himanshu Devrani, Jayant Haritsa, and Harish Doraiswamy. Query Rewriting via LLMs. *arXiv, 2502.12918*, 2025.

[5] Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, and Xinyu Wang. SlabCity: Whole-Query Optimization Using Program Synthesis. *PVLDB*, 16(11):3151–3164, July 2023.

[6] Cesar A. Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, 1997.

[7] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[8] Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[9] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. of ICDE*, page 209–218, USA, 1993. IEEE.

[10] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293, Berlin, Heidelberg, 2006. Springer.

[11] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. LLM-R2: A Large Language Model Enhanced Rule-Based Rewrite System for Boosting Query Efficiency. *PVLDB*, 18(1):53–65, September 2024.

[12] Jie Liu and Barzan Mozafari. GenRewrite: Query Rewriting via Large Language Models. *arXiv, 2403.09060*, 2025.

[13] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1275–1288, 2021.

[14] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[15] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2018.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[17] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule-based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 39–48, 1992.

[18] PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[19] Greg Rahn and contributors. tpcds-kit: TPC-DS benchmark kit with modifications and fixes. GitHub repository. Based on version 2.10.0 of the TPC-DS benchmark kit. Accessed: 2026-04-07.

[20] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[21] Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. QUITE: A Query Rewrite System Beyond Rules with LLM Agents. *arXiv, 2506.07675*, 2025.

[22] Zhaoyan Sun, Xuanhe Zhou, Guoliang Li, Xiang Yu, Jianhua Feng, and Yong Zhang. R-Bot: An LLM-Based Query Rewrite System. *PVLDB*, 18(12):5031–5044, September 2025.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2 edition, 2018.

[24] Transaction Processing Performance Council. TPC benchmark™ DS standard specification. https://www.tpc.org/tpcds/. Accessed: 2026.

[25] Transaction Processing Performance Council. TPC benchmark™ H (decision support) standard specification. https://www.tpc.org/tpch/. Accessed: 2026.

[26] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proc. of SIGMOD*, page 94–107, New York, NY, USA, 2022. ACM.

[27] Dongjie Xu, Yue Cui, Weijie Shi, Qingzhi Ma, Hanghui Guo, Jiaming Li, Yao Zhao, Ruiyuan Zhang, Shimin Di, Jia Zhu, Kai Zheng, and Jiajie Xu. E3-Rewrite: Learning to Rewrite SQL for Executability, Equivalence, and Efficiency. *arXiv, 2508.09023*, 2025.

[28] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th international conference on data engineering (ICDE)*, pages 1297–1308. IEEE, 2020.

[29] Xuanhe Zhou et al. LearnedRewrite: Official GitHub Repository. https://github.com/XuanheZhou/LearnedRewrite, 2022. Accessed: 2026-03-31.

[30] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. A learned query rewrite system using Monte Carlo Tree Search. *PVLDB*, 15(1):46–58, September 2021.