UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

An Inquiry into Database Query Optimisation with Equality Saturation and Reinforcement Learning

George-Octavian Bărbulescu

Churchill College

June 2023

Submitted in partial fulfillment of the requirements for the Master of Philosophy in Advanced Computer Science

Total page count: 60

Main chapters (excluding front-matter, references and appendix): 47 pages (pp 7–53)

Main chapters word count: 14381

Methodology used to generate that word count:

```
\newcommand{\wordcountcommand}{
%
\immediate\write18{texcount -1 -sum=1,1,1,1,1,1,1
-merge report.tex > main.sum }%
\input{main.sum}%
}

\wordcountcommand
```

Declaration

I, George-Octavian Bărbulescu of Churchill College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: George-Octavian Bărbulescu

Date: June 1, 2023

Abstract

Relational query rewriting involves rewiring a SQL query plan into a more competitive equivalent one. The ability to devise efficient query plans is paramount to the overall performance of the underlying database system. Henceforth, existing systems leverage decades of knowledge to perform SQL rewrite. Generally, mainstream databases rewrite query plans using white-box strategies, such as human-crafted heuristics. Two overarching limitations arise in this context. First, rewrite rules have inter-dependencies and may prevent future superior optimisations by obfuscating them. Second, finding the optimal rewrite order is NP-hard, with a search space exponential in the number of rules.

In this research project, we address these challenges by introducing *equality saturation* (ES) as a medium for building efficient SQL plans. Equality saturation transforms the rewrite ordering problem into non-destructive graph rewriting, where a rule will merely inject an alternative SQL plan without interfering with the original one. We show theoretically and empirically that equality saturation is a more competitive query rewrite scheme than current SQL planners. Moreover, this work extends equality saturation in two ways. First, we introduce a graph reinforcement learning system to guide the application of rules, showing it outperforms the state-of-the-art equality saturation solvers in a number of compiler domains, including query planning. Second, we analyse the ability to learn with graph representation learning and recurrent neural networks over *equality graphs* (e-graphs), the data structure used in ES to represent a congruence relation over compiler plans.

Acknowledgements

First and foremost, this project would not have been possible without the wonderful support of my supervisor Eiko Yoneki and my two co-supervisors, Taiyi Wang and Sami Alabed. I would also like to express my gratitude to Malcolm Scott for the hardware resources.

Contents

Chapter 1

Introduction

Databases (DBs) are intricate systems for storing and managing data. To manipulate the data, DBs allow users to submit queries. A query is a declarative statement through which the user specifies *what* data to fetch or manipulate. The DB system manages *how* the query is executed through a query plan. Rewiring a query plan is a fundamental problem in query optimisation. Query rewrite systems transform a database transaction into an equivalent and more efficient one. If executed effectively, the rewrite process can speed up the original transaction by orders of magnitude [70]. The rewrite system takes as input a database query and projects it into a logical plan. Subsequently, the plan is rewired based on pre-defined rules (e.g. remove redundant operators) with two overarching goals in mind. First, the system seeks to produce a query plan that improves upon the original execution time. Second, the devised plan has to be equivalent to the original.

Traditional query optimisers apply optimisations in a predefined order, with each rewrite operating on the output of its predecessor [4, 42]. For example, mainstream databases such as PostgreSQL [42] take a top-down approach to the task at hand. Starting from the query plan root, query operators are replaced with equivalent expressions as soon as they match a rule pattern. The inherent limitation of this approach is the potential to fall into a local optimum [70]. A brute-force approach to address the problem is to exhaustively search all possible partial rewrite orders and pick the best-rewritten form based on a cost metric such as latency. However, the rewrite order space grows exponentially with the number of available rules. To further exacerbate the issue, the rewrite time is also accounted for in the overall latency of the database transaction.

Henceforth, existing methods suffer from two main problems. First, encoding a substantial number of rewrite orders is difficult, given the significant space size. Second, finding the optimal rewrite order is proven to be an NP-hard problem [10, 40]. The problem can be linked to the *phase ordering* problem from the compiler domain. With the same reasoning in mind, traditional compilation systems can produce sub-optimal programs when an optimisation earlier in the chain obfuscates more superior future rewrites.

In this research project, we repurpose *Equality Saturation*, an algorithm proposed by Tate et al. [58] to address the phase ordering problem, a phenomenon in traditional compilation systems analogous to query rewrite ordering. Equality saturation simultaneously represents infinitely-many execution plans through specialised data structures coined *equality graphs* (e-graphs) [37, 36]. By efficiently encoding the space of all possible optimised versions of a query, the algorithm obviates the need to optimise ordering under strong assumptions. One such assumption is that the encoding (i.e., the e-graph) fits into memory.

We investigate the practicality of equality saturation through the lens of the state-of-the-art implementation coined *e-good graphs* (egg) by Willsey et al. [64]. The authors introduce solutions to speed up the algorithm and to make it domain-agnostic. However, applying egg in complex compiler domains (e.g. to rewrite queries) within limited space is a compromised solution. For example, encoding all rewritten versions of a database query in an e-graph within limited space is unrealistic due to infeasible memory requirements. In turn, this may prevent the algorithm from extracting the globally-optimal query plan.

An intuitive approach to tackle the said limitations is to *grow* the e-graph (i.e., to encode alternative plans) in such a way that the most competitive plans are encoded within the memory limit. However, this strategy is not generalisable across different programming languages, as it requires domain-based heuristics. To alleviate this problem, we direct our attention to reinforcement learning (RL) and investigate whether a learning-based approach can remove the dependency on the domain knowledge and whether such a method springboards good query optimisers. Inspired by previous efforts of introducing RL in equality saturation such as Omelette [52], we devise Aurora. Aurora is a query optimiser that interleaves RL and equality saturation to address the query rewrite problem. The proposed methodology interleaves graph representation learning methods and recurrent neural networks, which we coin *"spatio-temporal"* learning, to learn an RL policy over database query e-graphs. Concretely, our contributions are:

1. The recognition of theoretical and practical limitations of state-of-the-art equality saturation solvers when applied to the database domain.

2. The first formulation of the query rewrite problem as an RL-based equality saturation task, to the best of our knowledge.

3. The design and implementation of Aurora, a database query optimiser that leverages RL-based equality saturation. The methodology encompasses: a) a novel spatio-temporal RL algorithm for solving combinatorial problems over graphs; and b) an integer linear programming (ILP) formula for extracting query plans from e-graphs.

4. An extensive experimental evaluation of Aurora against the Omelette and egg rewrite systems on two grammar domains: the query rewrite language and mathematical simplification. We show that Aurora reduces the cost of query plans by

orders of magnitude compared to the baselines, while also providing competitive results across other programming languages.

Chapter 2

Background and Related Work

This chapter introduces the techniques required to contextualise the methodology of this project. The research hypotheses revolve around equality saturation as an optimisation medium for database queries. We first describe the act of optimising a relational database query. This helps to draw a bridge between query optimisation and term rewriting. Subsequently, we discuss equality saturation as a method to perform term rewrite. Before discussing the prominent literature from both worlds, we take a detour and introduce reinforcement learning. Finally, this chapter explores related work from query optimisation and equality saturation to show that intersecting the two is a promising research avenue.

2.1 Query Optimisation and Query Rewrite

A database transaction is a communication method between human or application actors and a database system. A Structured Query Language (SQL) transaction is a formal statement that specifies the data to be retrieved from the storage system. Queries are composed of operators in charge of filtering data, selecting specific parts of the data space, or performing calculations over data (e.g. sum).

To enable efficient execution of queries, database management systems (DBMSs) employ query optimisers. The query optimiser is in charge of finding correct execution plans for the submitted transactions, while also reducing costs such as system latency.

A fundamental part of query optimisation is query rewrite [40, 69, 10, 70]. The aim is to project the original query plan into a plan that exhibits better performance. Concretely, a query rewrite system alters a SQL query at the logical level (e.g. by discarding redundant operations, changing the order of operations) in a manner that keeps the output equivalent to the original one and reduces the latency. This problem is proven to be NP-hard [10, 40]. Figure 2.1 renders an example.

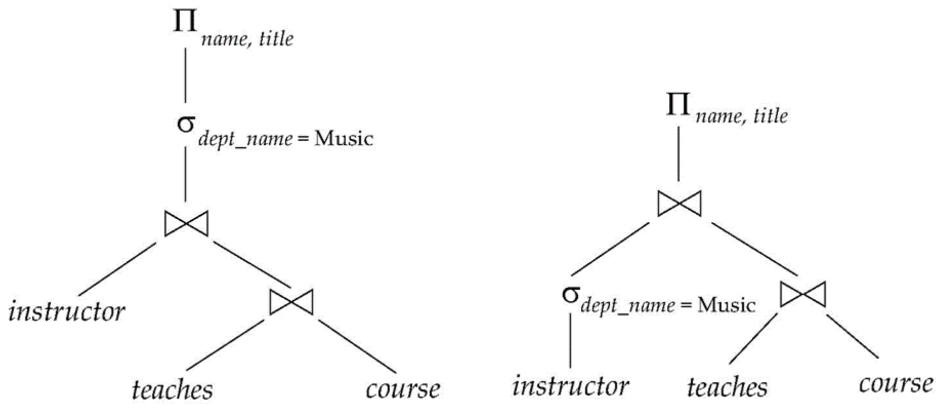


Figure 2.1: Logical query plan optimisation. Example based on [51].

Figure 2.1 shows an iconic plan rewrite step. In this case, the optimiser decides to push the filter on *"dept_name"* through a join operation. By rewiring the plan, the query optimiser estimates the top-most join operation will be faster, which leads to a decrease in the overall latency of the plan. Intuitively, the reason for higher performance is linked to the number of rows the system operates on. By filtering the table *"instructor"* early in the plan, the subsequent operations are executed on fewer data.

Another rewrite opportunity arises when dealing with nested sub-queries. The DBMS regards nested sub-queries as functions and expects them to return a set of values. The optimiser chooses between abstracting a sub-query into a temporary table and flattening it based on correlations. The optimiser's choice is reflected in the end-to-end latency of the query. The reason is that the former rewrite rule will render the latter inapplicable. An example is given later in the project in Figure 3.1

2.2 Equality Saturation

Query optimisers handle the rewriting problem by matching queries or patterns in the SQL plan with predefined rules. Their limitations arise from the order in which the rewrite rules are applied, which may lead to sub-optimal plans. Equality saturation is a methodology proposed in compiler literature to tackle the phase ordering problem [64]. This section investigates the moving parts of equality saturation and contextualises how it can be adapted to the query rewrite case.

2.2.1 Equality Graphs

Equality saturation employs equality graphs (e-graphs), which are data structures that efficiently represent congruence relations [36, 37]. E-graphs, initially developed for automated theorem provers (ATPs) [64], extend the union-find method proposed by Trajan et al. [57] to efficiently encode expressions in equivalence classes. A core property of the e-graph is the ability to close the equivalence relation under congruence (e.g. if $a \equiv b \rightarrow f(a) \equiv f(b)$).

Over the past decade, e-graphs have become ubiquitous in rewrite-driven compiler optimisations, as part of a technique coined equality saturation [58, 60, 61, 65]. Equality saturation receives an input program and stores it in an e-graph. Subsequently, the e-graph is expanded iteratively by applying rewrite rules over the program. An important remark is that growing the e-graph is a purely additive process - the rewrites solely add information instead of changing the original. Upon reaching *saturation*, the e-graph is deemed to represent all possible equivalent plans for the program.

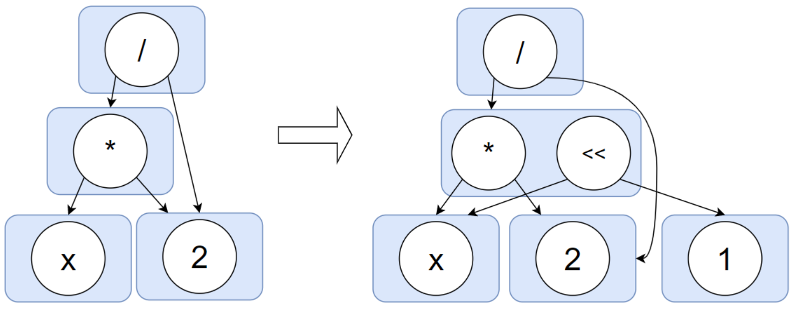


Figure 2.2: E-graph construction. The e-graph undergoes rewrite $x * 2 \rightarrow x << 1$

Figure 2.2 shows the construction of an e-graph for mathematical optimisation. The rewrite rule $x * 2 \rightarrow x << 1$ only adds information to the original expression $(x * 2)/2$. If there exists a rule such that $(x * 2)/2 \rightarrow x$, this rule can be subsequently fired over the e-graph to produce an optimal plan in terms of the number of operations. In a classic term rewrite system, the left-hand side expression $(x*2)/2$ would have been obfuscated by $(x << 1)/2$, thus leading to an ordering problem. An e-graph consists of e-classes (blue boxes) and e-nodes (circles). An e-class contains one or more e-nodes. Picking any e-node from an e-class is ensured to produce a correct and equivalent expression in reference to the original.

2.2.2 Algorithm

This section presents the anatomy of equality saturation as a method to perform graph rewriting. The high-level reasoning is adapted from [64] and rendered below:

Listing 2.1: Equality saturation pseudocode

```
def equality_sat(p:Program, rewrites:List, cost_fun:CostFunction):
    e:EGraph = egraph(p)

    while not e.is_saturated_or_limits():
        for rw in rewrites:
            for (subst, eclass) in egraph.match(rw.left):
                new_eclass = egraph.add(rw.rhs.subst(subst))
```

```
8                 egraph.merge(eclass, new_eclass)
9
10    return egraph.extract_best(cost_fun)
```

The algorithm in Listing 2.1 shows a bird's eye view of equality saturation. A new e-graph is generated from an input program p. The e-graph is iteratively grown by *additive* substitution. Concretely, if the left-hand side of a rewrite rule matches a pattern in the e-graph (line 6), new e-classes are generated to introduce the substitutions (line 7). The *merge* oracle blends two equivalent e-nodes into the same e-class under the congruence constraint (line 8).

Once the e-graph is saturated, the optimal version of the input expression is extracted from the e-graph based on a user-defined cost function (line 10). The classic extraction approach is a greedy bottom-up traversal of the e-graph. However, certain applications require more advanced techniques such as solving Integer Linear Programs (ILPs) [64]. This is also the case in the SQL domain and we dedicate an entire section to our proposed cost model in Chapter 3.

The advantage of leveraging an e-graph to perform rewriting is twofold: 1) it helps minimize the memory needed to store a potentially infinite number of equivalent plans, and 2) it provides a medium for tackling the phase-ordering problem in a more efficient way than exhaustive search. A brute-force method such as a breadth-first search across all possible expressions would lead to combinatorial explosion.

The algorithm is domain-agnostic and the specifics of the *rewrites* and the program p are solely dependent on a programming language. This raises the question of whether the query rewrite problem can be translated into an instance of equality saturation. We remark that SQL dialects can be translated into relational algebra, a formal language for specifying SQL which comes equipped with an extensive range of algebraic equivalence transformations [24, 10]. This realisation serves as the foundation for our research hypotheses, which seek to efficiently adapt domain-agnostic equality saturation for query rewrite.

2.3 Reinforcement Learning

Reinforcement Learning (RL) encompasses a class of techniques for optimising sequential decision-making processes. Generally, RL is applied to maximise the return in a Markov Decision Process [56]. Markov Decision Processes (MDPs) are instruments for characterising sequential decision-making problems over a discrete time dimension [44]. The decision-making process is associated with an agent that is able to act over a Markov chain [38], an environment where the agent's future state depends only on its previous state and the action taken. This is known as the Markov Property [12], and it reduces the complexity of the model by removing the dependency on historical state-action pairs to quantify the outcome of an action.

Formally, an MDP is a 4-tuple (S, A, T, R) which describes the following:

1. S: represents the space of possible states.
2. A: describes the set of possible actions an agent can take in the environment. The choice of an action will transition the agent into a new state.
3. T: characterises the dynamics of the environment and it is known as the transition function in RL parlance. The transition function acts over the state-action space $T : S \times A \rightarrow S$, outputting the new state s' when action a is taken in state s. The dynamics of the system can be either probabilistic $p(s'|s, a)$ or deterministic $T(s, a) = s'$.
4. R: represents the reward signal. The transition to a new state is succeeded by an immediate reward. This quantifies the local benefit of taking an action a in state s and landing in state s'. $R(s, a, s')$ is the signal that guides the optimisation process.

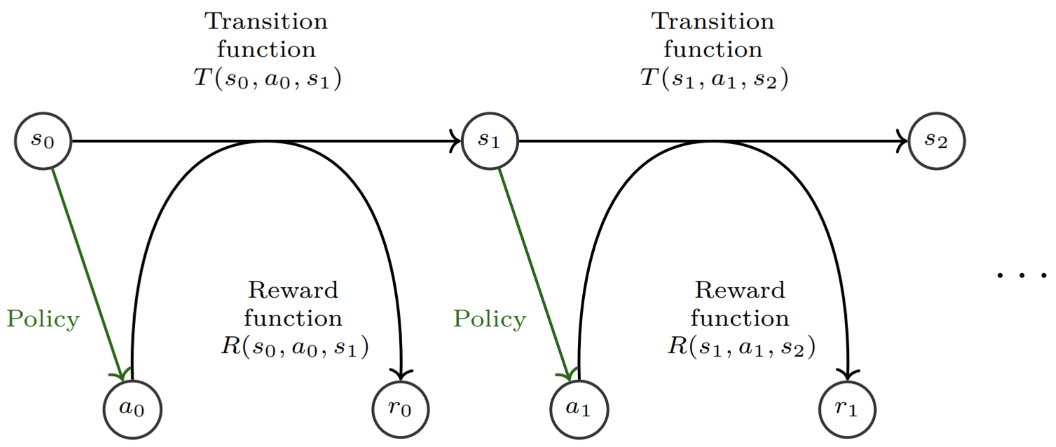

Transition function $T(s_0, a_0, s_1)$
Transition function $T(s_1, a_1, s_2)$
s_0
s_1
s_2
Policy
Reward function $R(s_0, a_0, s_1)$
Policy
Reward function $R(s_1, a_1, s_2)$
. . .
a_0
r_0
a_1
r_1

Figure 2.3: The sequential workings of a Markov Decision Process. Illustration taken from [11].

The goal of RL is to learn a policy, denoted as π, that maps the states of an MDP to actions in order to maximise the cumulative reward of the agent in the environment. In contrast to a greedy approach, RL methods operate over the expected rewards rather than the immediate ones. A popular method to find the policy π is Q-learning [63]:

$$Q(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')|S_t = s, A_t = a], \gamma \in [0, 1) \tag{2.1}$$

In Q-learning, the function over a state-action pair $Q(s, a)$ is coined *Q-value* and it defines a proxy over the expected cumulative reward that an agent can obtain by taking an action in a specific state. Formally, the agent follows policy $\pi(s_t, a_t)$, where action a_t corresponds to the maximum state-action value $Q^*(s_t, a_t)$. In order to estimate state-action values, Q-learning algorithms leverage the Bellman equations [56, 2] in an iterative fashion, with the guarantee that $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$.

In practice, agents may need to memorize very large state-action spaces, compromising the ability to infer a competitive policy within reasonable memory and time. Artificial neural networks (ANNs) tackle this by approximation, leading to the following compromise: $Q(s_t, a_t; \theta) \approx Q^*(s_t, a_t)$. Intuitively, the aim is to produce Q-value estimates to generalise across unseen state-action pairs from the same distribution, by learning parameters θ. This is known as Deep Q-learning (DQN) [34].

As opposed to estimating the value functions to infer a policy, a more intuitive approach using ANNs is to directly learn the policy. Policy gradient methods directly optimise the policy $\pi(a_t|s_t)$, instead of the cumulative reward of a state-value pair. Furthermore, policy gradient methods automatically enable the agent to explore the environment by defining probability distributions over the action space. Deep learning is formally interleaved to define the stochastic policy as follows:

$$\pi_\theta(a_t|s_t) = p(a_t|s_t, \theta) \tag{2.2}$$

The parameters of the deep model θ are optimised to alter the probability distribution based on the agent's interactions with the environment. To this end, the probabilities of actions that lead to higher rewards in the environment are increased to the detriment of actions with meagre yield.

2.3.1 Proximal Policy Optimisation

Proximal Policy Optimization (PPO), introduced by Schulman et al. [47], is a policy gradient method that trains *actor-critic* ANNs to infer a policy. PPO addresses the same research question as its predecessor, Trust Region Policy Optimisation (TRPO) [46]: how to guide the policy improvement in a constrained fashion to prevent negative performance drifts? The most popular approach in the literature is to optimise the policy over a clipped surrogate function as follows:

$$\mathcal{L}_{\text{clip}}(\theta) = \mathbb{E}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\hat{A}_t, \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{2.3}$$

The advantage factor $\hat{A}_t$ is a proxy for the state-action pair at time-step t. Intuitively, this quantifies the benefit of taking action a in state s, and it's computed based on the *critic* network which maps state observations to predicted values. The *clip*$(\cdot)$ oracle constrains the probability ratio term $\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ in the range of $[1-\epsilon, 1+\epsilon]$. In simple terms, the change in action probabilities should not be steeper than a hyper-parameter ϵ. This leads to a more conservative and therefore stable training routine. By optimising the objective function in Equation 2.3, the *actor* network, which maps observations to actions, is iteratively pushed towards the optimal policy.

2.4 Related Work

This section investigates the prominent research from the database community linked to query rewrite. Subsequently, a range of compiler optimisation literature that improves upon the classic equality saturation methodology is explored. The aims of this research work are not only to quantify whether equality saturation is a good medium for query rewrite, but also to research new techniques to improve upon domain-agnostic equality saturation with reinforcement learning. The intention of the latter goal is to make equality saturation memory and time efficient.

2.4.1 Literature on Query Rewrite

Query optimisation has a long history of research in the database community. The proposed methods can be clustered into two areas. The first class of methods within the taxonomy encompasses techniques that jointly optimise the logical and the physical facets of a query [62, 4, 35]. The second cluster of methods proposes a *stratified* approach to query optimisation [27, 48, 40], which splits the process into logical rewrite and physical rewrite.

Recently, machine learning techniques have been introduced into the query optimisation pipeline at different critical points. For instance, a deluge of work addresses cost estimation [55, 32] to account for the meagre approximation tools available in mainstream optimisers. On a parallel front, deep learning solutions for the *join ordering* problem have emerged. The join ordering problem coins that the execution latency of a query involving multiple tables is greatly impacted by the order in which the tables are joined. This is generally an NP-hard [22] problem as the number of join orders is exponential to the number of tables involved. The problem has been addressed in a promising fashion through techniques such as RL [30, 66]. However, much of this work neglects other logical rewrite opportunities. While current methods can efficiently estimate costs and propose competitive join orders, the field is scarce in terms of rewriting complex queries. The reason is two-fold: 1) the space of possible rewritten queries can be enormous, and 2) the ordering problem is very difficult to learn.

Of particular importance, we identify the work of Zhou, Xuanhe, et al. [70]. The authors attack the logical query rewrite problem by employing RL [56]. The proposed method seeks to guide the application of relational rewrite rules by learning to estimate the performance of rewritten queries using Monte Carlo Tree Search [8]. Orthogonal to this work, WeTune [62] brings a refreshing perspective on the field through a pipeline capable of automatically generating SQL rewrite rules.

2.4.2 Literature on Equality Saturation

Re-purposing equality saturation for query rewrite is orthogonal to the work of Zhou, Xuanhe, et al. [70], who seek to learn over the space of all query rewrite orders. Instead, we propose to reduce the search space by learning over the query e-graph space. To this end, we direct our attention to two prominent strategies from compiler literature that improve upon equality saturation, which will serve as the baselines of our experimental analysis.

egg

The e-good graphs (egg) framework is the state-of-the-art library for equality saturation. It revitalises the classic methodology by speeding up the algorithm. The modern library proposed by Willsey et al. [70] is domain-agnostic and exhibits asymptotic latency improvement over previous equality saturation solvers [58]. As a concrete example, the authors introduce *rebuilding* as a novel mean to amortise the costs associated with restoring e-graph invariants (line 8 in Listing 2.1). The technical details associated with the egg framework are outside the scope of this work. Of particular importance is that egg applies the input rewrite rules in sequential order. This is analogous to constructing the e-graph by iterating over the list of rewrite rules one by one.

Omelette

Omelette [52] extends the *egg* framework by treating the application of rules as a graph combinatorial problem and solving it with RL. Informally, the intuition behind Omelette is to guide the application of rules within equality saturation, in order to encode as many competitive plans in the e-graph as possible within a memory limit. The aim is to strike a balance between the best compiler plan that can be extracted from an e-graph and the size of the underpinning e-graph. The author shows that a *wise* application of rewrite rules can minimise the growth of the e-graph data structure, and consequently, the memory requirements associated with equality saturation, while also ensuring competitive results linked to the quality of the rewritten expression.

Discussion

By now we have investigated research trends from two disparate worlds: query optimisation and equality saturation. In the next chapter, we motivate our work through concrete query rewrite examples and show how equality saturation is a natural fit for the problem. Subsequently, we discuss the limitations of the default equality saturation algorithm in complex language domains, such as the database domain.

Chapter 3

Aurora: RL-based Equality Saturation for Query Optimisation

This chapter presents the research methodology for this project. The first section delves into the open problems associated with query rewrite. Simultaneously, we investigate the failure points of equality saturation in practical real-world systems.

Following the discussion, we present the research hypotheses. To verify these, we introduce Aurora, a novel reinforcement learning pipeline for database-oriented equality saturation. The proposed methodology covers grounds from graph representation learning to recurrent neural networks with the aim of automating e-graph construction. While the techniques presented revolve around the query optimisation scenario, they can be applied to any programming language designed for equality saturation. This makes the methodology comparable to domain-agnostic equality saturation solvers such as egg and Omelette.

3.1 Challenges in Query Rewrite

Rewiring SQL queries into more efficient equivalent expressions is an NP-hard problem [40, 10]. The query engines in popular database management systems (e.g. PostgreSQL) are intricate mechanisms. In essence, a transaction engine takes a query written in a declarative language (e.g. *"select * from table t"*) and turns it into an executable plan. Generally, the query is projected into a logical plan and optimised by following a rewrite-based routine. The plan comprised of operator nodes is destructively rewired based on hand-crafted rules (e.g. perform filtering foremost in the plan). The application of rules commonly follows an ordering scheme (e.g. arbitrary, heuristic) [70]. In turn, this can lead to suboptimal executable plans, as rewrite rules may have inter-dependencies. Concretely, destructively applying a rewrite rule over the SQL plan renders other possible optimisations infeasible, if the pattern used for matching is discarded by the initial rule.

To build our intuition, we refer to a study case provided by the authors of [70], who investigate the inner workings of PostgreSQL. PostgreSQL facilitates database transactions in a top-down manner. The engine traverses the logical query plan starting from the root and performs local rewrites iteratively.

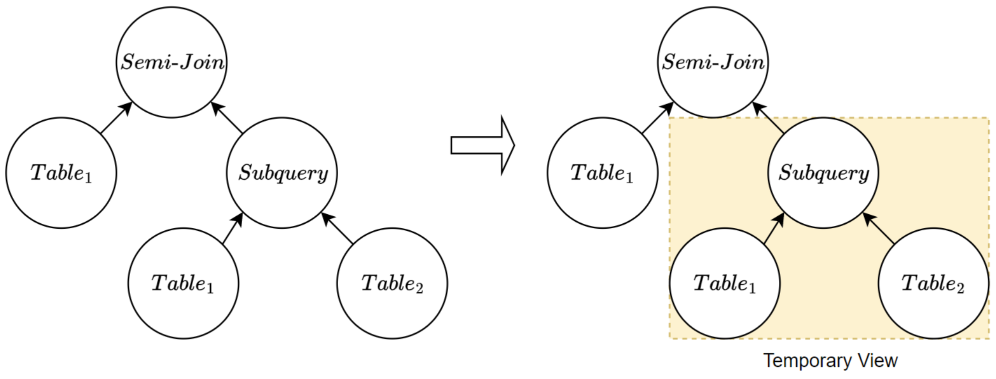


Figure 3.1: The original SQL plan (left) is optimised in a top-down fashion by creating a temporary view over the subquery (right).

Figure 3.1 renders an example SQL query for which a top-down optimiser finds a sub-optimal plan. The original plan tree is representative of SQL templates such as *"select _ from $table_1$ with condition over (select _ from $table_1$, $table_2$ where _)"*. Once the optimiser reaches the *Subquery* node in the plan, it naively creates an inline view. Consequently, further optimisations are disabled over the view. By disregarding the characteristics of the leaf nodes, a top-down application of rewrite rules will not take advantage of the correlation between the subquery and the outer query (note the same relation - $Table_1$ - is scanned twice). Ideally, the subquery is pulled up as a join to prevent a redundant scan operation. In practice, this simple optimisation reduces the execution time by a factor of 600 as per the original authors [70].

Having established how mainstream query optimisers fall into local optima due to the rewrite ordering problem, it is paramount to question the frequency of this scenario in practice. After all, if most analytical queries are trivial to optimise, then the overhead associated with suboptimal plans can be rendered insignificant. First, suboptimal plans devised by mainstream optimisers such as PostgreSQL can be orders of magnitude slower than their optimal counterparts [26, 69]. This is far from an *"insignificant overhead"*. Second, we argue the volume of non-trivial queries is large. On one hand, the query engine has to optimise human-written queries. The SQL queries devised by human actors are characterised by intuitive patterns, and the decades of hand-crafted rules generally address these [14, 15, 13, 25, 62, 48]. On the other hand, there has been an explosion in the number of queries generated by systems rather than human actors in recent years. For instance, in the context of web development, object-relational-mapping (ORM) allows database transactions to be generated by the underpinning web application [62].

We analyse a query template from GitLab produced by [62]: *"select id from notes where*

_ and id in (select id from notes where _)". The query fetches the *id* values from the table *notes*, filtered by two consecutive conditions. Once again, PostgreSQL does not leverage the correlation between the subquery and the outer query to rewire the logical plan into a single select statement over the *notes* table. The importance of the result is two-fold: 1) it shows that mainstream DBs may fail even for simple queries where there is no human guidance to distil counter-intuitive patterns; 2) it points to the fact that system-generated queries are treated in a sub-optimal fashion by existing engines.

Summary. This section has shed light on several limitations exhibited by existing systems. First, we have investigated how top-down query rewrite schemes get trapped into a local minimum. An orthogonal problem is how to encode a substantial number of rewrite orders, or otherwise SQL plans [70]. Second, we point towards the fact that a large corpus of queries may actually be non-trivial to optimise. We deem equality saturation as a promising research avenue, as the facets of our problem tend to be inherently addressed by its methodology. However, existing equality saturation solvers come with their own problems in the SQL domain, as we argue in the next section.

3.2 Challenges in Applying Equality Saturation

Equality saturation (ES) starts from the promise to address the ordering problem in classic rewrite-based systems. Different from term rewriting systems, which substitute blocks from the original expression, the ES methodology proposes the employment of e-graphs to encode all rewritten versions of the input expression. The e-graph plays a crucial role within ES, facilitating various auxiliary tasks such as maintaining congruence (see Section 2.2 on equality saturation). However, if equality saturation is applied without careful consideration, the e-graph can become a bottleneck.

Willsey et al. [64] render a scenario in which equality saturation is characterised by non-termination, where a purely *destructive* approach would terminate. In this context, we denote destructive any rewrite system in which the fired rules replace patterns from the original expression. The condition of non-termination is linked to an unbounded e-graph expansion, one that never saturates.

The second concern raised in literature is the prominence of the framework to run into *"explosions"*. We coin an explosion as a soar in the number of nodes the e-graph has in the aftermath of a meagre rule application. As a concrete example, Yang et al. [65] investigate this dimension in the context of tensor graph optimisation, showing that rewrite rules can quickly grow the e-graph to the upper limit of $50k$ (thousands) nodes they set. The consequences of a large e-graph are twofold: 1) the data structure may not fit into memory in complex domains where explosions are frequent; and 2) the routines required to perform equality saturation (e.g. extraction, restoration) become computationally inefficient.

The limitations raised above are intrinsic to the default equality saturation algorithm which grows the e-graph until all rewritten versions of the original expression are encoded. This leads to infeasible computation runtime and unrealistic memory requirements. We remark that in prior work [65, 64], equality saturation is applied for offline optimisation. For instance, optimising the computation graph of a machine learning model is done before the model is used for inference.

A less explored realm is equality saturation for online optimisation. We define *online* within the boundaries of this research project, as the application of equality saturation in a system that provides real-time feedback, such as a database query engine. In this scenario, the overhead associated with growing the e-graph and extracting the optimal expression is included in the end-to-end latency of a query. Furthermore, as e-graphs can grow to thousands of nodes, it is cardinal to reflect on the memory overhead associated with optimising database queries concurrently. These dimensions are explored in Chapter 4, which encompasses our experimental evaluation.

Summary. We have raised concerns related to the practicality of equality saturation for online systems. Different from the offline case, where compromises can be made in terms of optimisation latency, the database domain is rather unique through the fact that equality saturation has to provide latency improvements proportional to the optimisation overhead.

3.3 Overview & Hypotheses

To jointly address the challenges underpinning query rewrite and the concerns associated with applying the state-of-the-art equality saturation for this NP-hard problem, we propose Aurora, an RL pipeline that learns over the space of SQL e-graphs.

By re-purposing equality saturation for query optimisation, we aim to address two problems. First, existing methods cannot efficiently represent a large space of rewrite orders. Second, finding the optimal rewrite order across disparate database management systems (e.g. PostgreSQL, Oracle [16]) is computationally infeasible. Concretely, we hypothesize and prove the following:

1. *H1: Aurora can rewrite complex database queries to significantly reduce end-to-end latency*
2. *H2: Aurora is generalisable across different database management systems*

Simultaneously, by re-introducing the rewrite problem in equality saturation and solving it with RL we mitigate the constraints associated with the optimisation latency and the stringent memory requirements. Intuitively, we transpose the problem from RL over the space of rewrite orders to RL over the space of e-graphs, in an attempt to reduce the complexity of the problem. To this end, we further hypothesize and prove the following:

1. *H3: Aurora is able to find lower cost solutions than the leading methods for equality saturation such as Omelette and egg*
2. *H4: Aurora provides unique advantages over other RL-based equality saturation techniques in terms of convergence and these advantages generalise across different grammars*

3.3.1 System Architecture

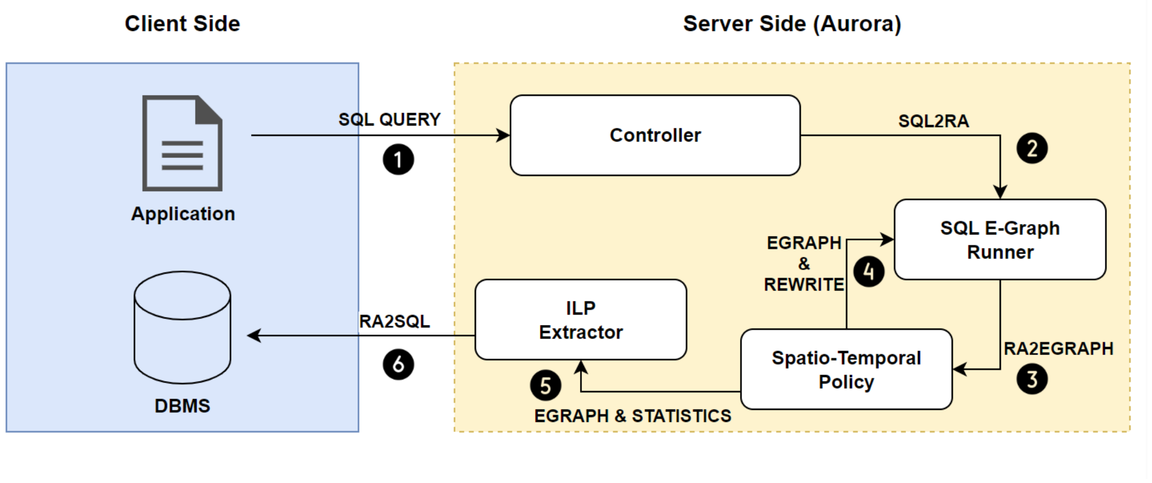


Figure 3.2: A 6-step schematic workflow of Aurora

Figure 3.2 renders the data workflow in Aurora. The system's actions are clustered into five overarching steps, with an optional sixth step that shows how the framework can be used as a software as a service (SaaS) for mainstream databases (e.g. PostgreSQL).

- *Step 1:* Aurora receives a relational database query from an application system or from a human actor. The *controller* entry point handles administrative routines such as opening database cursors and preparing the RL pre-trained agent for inference.
- *Step 2:* The SQL query is parsed into an *extended* relational algebra (RA) expression, with the caveat that the logical plan can include physical operators (e.g. hash-join, merge-join). The motivation behind the design is to jointly rewrite the query and to set flags for physical execution.
- *Step 3:* The logical plan is injected into an e-graph, the data structure used to represent a congruence relation over query plans.
- *Step 4:* The equality saturation policy chooses a rewrite rule from a fixed range until an optimisation goal is met. The pre-trained actor guides the expansion of the e-graph. This is analogous to enumerating query plans.
- *Step 5:* The e-graph and the statistics from the database catalogue (e.g. cardinality) are sent to an *extractor* module. The module extracts the best query from the e-graph according to a cost model. The extraction procedure is formulated as a

constrained optimisation problem over integer decision variables. The details are explained later in Section 3.5.

- *Step 6:* The extracted relational plan can be converted back into a SQL dialect, for instance, to be used in mainstream databases such as Oracle and PostgreSQL. This final step is optional in Aurora. The reason is that the framework acts as an end-to-end SQL optimiser with access to direct execution.

To address the limitations of existing equality saturation solvers, the e-graph construction is guided with RL (steps 3 and 4 in Figure 3.2). A pre-trained RL policy reasons over the subspace of e-graphs, deciding which rewrite rule to apply within the constraints of the system. The constraints are linked to the size of the e-graph and the planning latency.

The *SQL E-Graph Runner* is a specialised component that manages the e-graph, including the restoration of e-graph invariants (see Section 2.2 on equality saturation). The execution medium for the queries and the database management system (DBMS) in which Aurora resides by default is Risinglight [45], a Rust-based online analytical processing (OLAP) system. As an open-source contribution, we extend Risinglight to bridge our research work on graph representation learning and reinforcement learning with the Rust-based DBMS. The motivation for Risinglight as a DBMS is the smooth integration with egg, which is also written in Rust.

The SQL e-graph is defined over the operator clusters from Table 3.1. Classic algebraic operators, such as `PROJECT` and `FILTER`, are interleaved with physical proxies for operators such as `HASH-JOIN`. This allows for joint optimisation over the space spanned by relational query rewrites (e.g. pushing a filter through a join operation) and physical optimisation (e.g. replacing a join operator with a merge-join).

Operator Cluster	**Examples**
RA logical operators	`PROJECT, FILTER, JOIN`
Mathematical predicate operators	$+, -, \%, *$
Boolean predicate operators	`AND, OR, XOR, NOT`
Physical operators	`HASH-JOIN`
Aggregation functions	`SUM, COUNT, AVG, MIN, MAX`

Table 3.1: Example operators in the extended relational algebra (RA) language

The rewrite rules over the SQL e-graph are extended from [45] and [43]. A subset of the implemented rewrites is rendered in Table 3.2. We cluster the SQL rewrite rules based on the type of optimisation they bring to the query plan into physical rules, relational rules, and mathematical and Boolean rules. As an example, mathematical rules help with simplifying mathematical expressions.

Rewrite Cluster	**Examples**
RA logical rules	$filter(condition_1, filter(condition_2, table) \rightarrow filter(condition_1 \wedge condition_2, table)$
Mathematical predicate rules	$a + 0 \rightarrow a$
Boolean predicate rules	$a \wedge (a \wedge b) \rightarrow a \wedge b$
Physical rules	$join \rightarrow merge\text{-}join$
Join ordering rules	$join(a, b) \rightarrow join(b, a)$

Table 3.2: Example rewrite rules over the extended RA language

3.4 Reinforcement Learning for Equality Saturation

Section 3.2 has explored the challenging facets of applying equality saturation in complex language domains. The current state-of-the-art solvers are compromised solutions that can run into infeasible memory requirements and ignore the optimisation latency.

To address these problems, we introduce reinforcement learning in equality saturation to guide the e-graph construction process, which becomes a constrained optimisation problem. For a given input expression, the goal is to encode in the e-graph as many competitive equivalent expressions as possible within a limited time and memory budget.

The following sections expand on how equality saturation in the query rewrite domain can be formulated as an RL task. It is important to note our techniques are generalisable across any programming language and we prove this as part of the experimental analysis (Chapter 4) by optimising mathematical expressions.

3.4.1 Problem Formulation as Markov Decision Process

This section formalises equality saturation for query rewrite through the lens of a Markov Decision Process (MDP). As discussed in Section 2.3, the MDP is a common formalism for learning sequential decision-making.

State

Aligned with previous work from the realm [52], we use the e-graph as the state in our MDP model. Before delving into the specifics of our encoding procedure over the space of all possible SQL e-graphs, we first describe how a query becomes an e-graph.

The first step is to project the input query into the extended RA language Aurora uses. For instance, the SQL query *"select * from (select * from t where c2=True) where c1=True"* is rewritten by the query planner as $\pi(\sigma_{c1}(\sigma_{c2}(t)))$.

First, turning the original query into a relational form reduces the complexity of the original SQL dialect. Furthermore, relational algebra can be generalised across different database management systems (DBMSs). Unlike off-the-shelf query engines that leverage hand-crafted heuristics and white-box knowledge to optimise a particular dialect (e.g. PostgreSQL, Oracle [28]), the proposed technique can be integrated within any relational query engine.

Second, devising rewrite rules over relational algebra is a simpler task than rewriting a SQL dialect, as RA benefits from a solid theoretical foundation [10, 37].

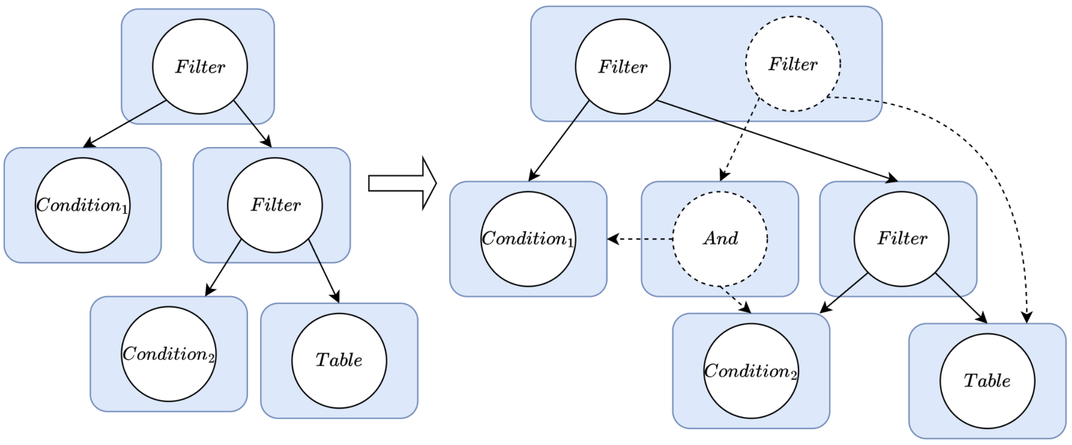


Figure 3.3: Example e-graph for RA expression $\pi(\sigma_{c1}(\sigma_{c2}(t)))$. By using the rewrite rule from Table 3.2, the SQL e-graph adds the equivalent plan $\pi(\sigma_{c1\wedge c2}(t)))$. The circles represent e-node operators and the blue boxes represent equivalence classes

The optimisation step shown in Figure 3.3 is a classic procedure in query planners. As discussed in Section 2.2, one advantage of the e-graph is the ability to encapsulate multiple semantically equivalent query plans, instead of destructively applying optimisation rules over the original plan. The transformation above merely adds information to the e-graph with an equivalent flat plan (note the filter operation is merged). In a destructive scenario, the original plan would have been compromised.

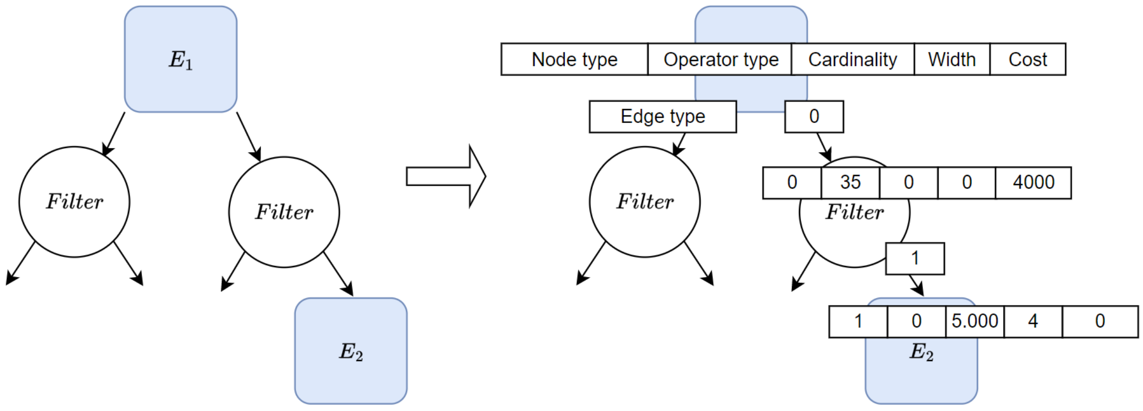


Figure 3.4: The proposed encoding scheme over the SQL e-graph space. The encoding scheme covers both the nodes as well as the edges in the e-graph. The underlying database system is queried for statistics such as cardinality, width, and cost estimates.

We take a slice from the rewritten e-graph rendered in Figure 3.3, which shows the top-most equivalence class (e-class $E1$), and two `FILTER` operators. We first discuss the node features, rendered in the diagram as a 5-tuple. The *node type* is an attribute intrinsic to the e-graph data structure. A node in the e-graph can either be an equivalence class (e-class) or an e-node. The e-class is a pointer for a set of e-nodes representing equivalent operators (e.g. `FILTER, JOIN`) from the query language. To distinguish between an e-class and an e-node we use a one-hot encoding to mark the type.

The *operator type* is the core component of the encoding scheme. After all, the rewrite actions over the e-graph are solely facilitated by the presence of certain operator patterns (e.g. a filter operation before a join operation). An intuitive approach is to one-hot encode the operators, which we show empirically to be expressive enough. The number of operators spanned by the extended RA language is 64. Alternative methods include a binary encoding over the operator range, in order to reduce the size of the feature vector to 7 features.

Finally, we add encoding support for statistics from the target database management system. There are three metrics we append to the node encoding. If the node is an equivalence class, then the feature vector stores the *cardinality* (number of rows) and the *width* (the number of columns/data bytes of a tuple) returned by its e-nodes. It is important to make a distinction here between e-classes and e-nodes. The e-nodes are executable operators (e.g. `FILTER`) and they can only be described through cost estimates. In the context of an e-graph, an e-node will never point to other e-nodes, but to child e-classes. To estimate the cost of an operator, we merely need to know the size of the output of each of its children. We remark that e-classes can be described through cardinality and width, regardless of which e-node is executed from that equivalence class. After all, if two e-nodes belong to the same e-class but yield different outputs, how can they produce semantically equivalent plans?

The second encoding scheme is applied over the edges of the e-graph. Once again, we use a one-hot encoding over the type of an edge. If the source of an edge is an e-class, then this establishes an e-class to e-node connection and we flag the first index as 1. Otherwise, we flag the second index that renders an e-node to e-class edge as 1 and set the former flag to 0.

In contrast to Omelette, we do not encode the rewrite rules, the terminals, and the type of non-operator e-nodes such as scalars (e.g. 0). There are no variables in the query scenario as all operations solely occur over tables and columns. While the encoding in Omelette increases expressiveness, we argue it may lead to generalisation problems. Furthermore, in the DB case, all parts of the e-graph can be solely described through DB statistics.

Action & Transition Function

The action space in the SQL e-graph construction MDP includes 37 actions. There are 35 actions defined over the SQL rewrite rules from Table 3.2.

The rewrite-based actions include multi-rewrite clusters as well as individual rewrite actions. For instance, one of the multi-rewrite actions we introduce looks at SQL predicate optimisation. This cluster encompasses 36 mathematical and Boolean rewrites, which are applied sequentially over the e-graph when the action is chosen by the agent. The reason we introduce multi-rewrites, rather than having individual rewrite actions, is to reduce the complexity of the combinatorial problem the RL agent has to solve. If we explode the said cluster into individual rules, the agent has to reason over double the number of choices at each iteration. The remaining 34 actions are split across join rules, plan rules (e.g. pushing a filter through a join operation), and physical optimisation rules.

The final two actions are specific to the constraints of our combinatorial problem. One of the open challenges in applying equality saturation to complex language systems is the *explosion* of the e-graph data structure. First, the e-graph is too big to efficiently apply rewrite rules over it and store it within limited memory. Second, the overhead associated with extracting the most efficient SQL plan from the e-graph is too high (see Section 3.2). In both cases, ES must be treated carefully, as latency improvements come at a planning cost.

To address these limitations, we introduce a *Reset* action and a *Stop* action. The *Reset* action extracts the optimal SQL plan found so far from the e-graph data structure. The aim is two-fold: 1) to induce a heuristic-like effect during e-graph construction, and 2) to reduce the size of the e-graph while maintaining opportunities for future optimisation. The *Stop* action is another non-trivial action the agent has to reason about. The RL policy is given the agency to stop an e-graph construction episode and extract the optimal SQL plan at the current timestep.

Reward Signal

Considerably the most difficult part of defining an MDP over a complex system is describing a robust reward signal. After all, the reward function is the main driver of the RL optimisation process. The reward quantifies how good the action the agent took in the environment is for the optimisation process. Given the large combinatorial space spanned by the defined action set and the e-graph operators (see Table 3.1), we define a *dense* reward to improve convergence.

Crafting a reward signal for equality saturation is non-trivial. The main goal of equality saturation is to transform an original SQL plan into an optimised one. In this context, we define an *optimised* SQL plan as a plan with lower execution latency than the original. As it is infeasible to extract and execute a SQL plan from the e-graph to record the latency at each timestep, we rely on a cost model. The specifics of the cost model are rendered in Section 3.5. For now, we assume the estimated cost (Figure 3.4) reflects the execution latency.

In terms of positive returns, the reward at each timestep is a function between the previous and the newly extracted plan cost. The most basic function outputs the absolute value of the difference between the two costs. This is a naive approach, as two equivalent plans can have largely different costs (e.g. an optimal join order versus a sub-optimal one). Alternatively, the reward function can look at the relative improvement between rewrites. This is also problematic when dealing with massive cost improvements in one step (e.g. from millions to thousands) and tiny cost improvements in subsequent steps. In our empirical analysis, we unveil a flat reward works well in practice, agnostic of the scale of the cost improvement. While this will qualify cost improvements of different magnitudes into the same reward *"bucket"*, the properties of the e-graph enable us to follow a sub-optimal reward signal without losing too much optimisation potential in an episode. After all, any rewrite action will merely add an equivalent plan rather than destructively change the SQL plan into a sub-optimal one. An argument can be made that having a flat reward signal will lead to meagre results under the constraint of a node limit. This is another reason for which we introduce the *Reset* action.

In terms of negative rewards, the agent is penalised when taking actions which are either saturated (i.e., don't add new nodes) or lead to quick e-graph explosions. If a rewrite brings little improvement but it appends a large number of nodes to the e-graph, then the agent is penalised according to the memory overhead.

3.4.2 Spatio-temporal RL

After formalising the MDP, the next step is to establish the RL methodology and the agent's deep learning architecture. We extend Proximal Policy Optimisation (PPO) to address the e-graph construction problem. See Section 2.3 for a brief introduction to RL methods and PPO. Choosing PPO is motivated by its conservative training routine, making it an ideal candidate for newly-explored environments, such as the e-graph construction MDP.

We attack the problem from two parallel fronts. First, we re-purpose PPO to tackle graph reinforcement learning by employing an *attention* encoder for learning representations over graphs. Second, we model the temporal link between sequential e-graphs by leveraging a recurrent neural network [33]. We theorize that by modelling the temporal link between a rewritten e-graph and its predecessor, the RL system captures the additive nature of the transition function. This is based on the fact that certain substructures of the e-graph will remain constant across the MDP episode (see Figure 3.3). The architecture of the agent is rendered in Figure 3.5.

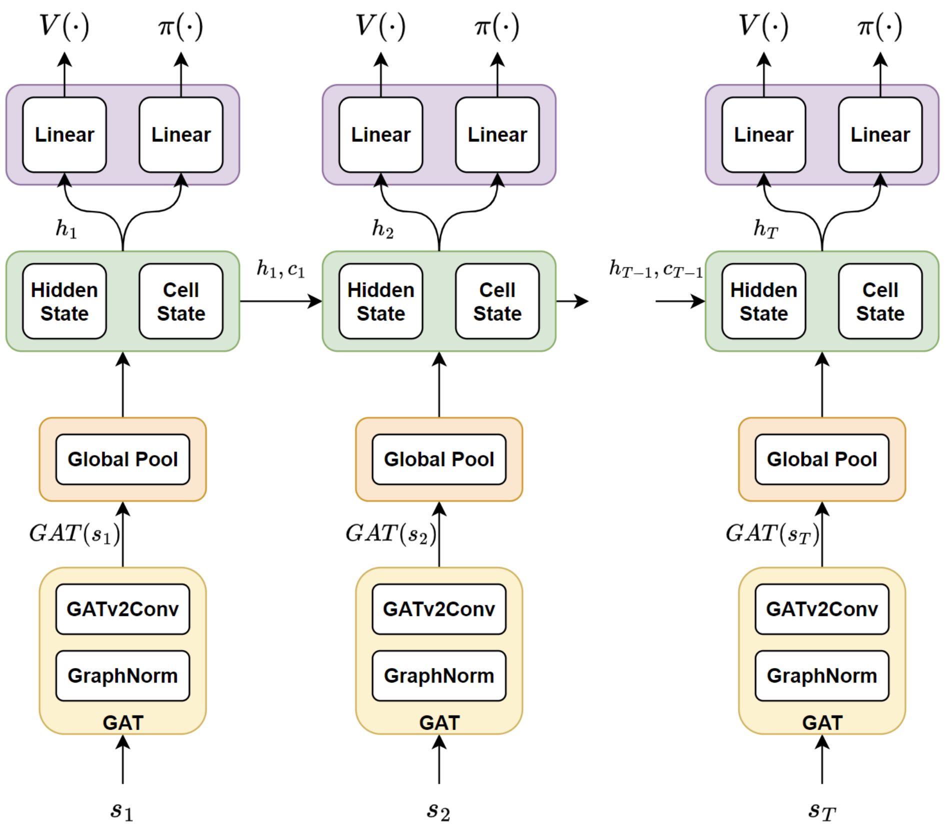


Figure 3.5: The architecture of the proposed deep RL pipeline

The deep model takes as input the state (i.e., the e-graph) at a particular point in time s_t and combines the trajectory modelled so far with the current encoding over the state. The first part of the deep model encodes the e-graph structure into a latent vector ready to be passed to a recurrent neural network. To achieve this, we employ tools capable of learning representations over graph-structured data.

Learning the Spatial Representation

In equality saturation, the e-graph serves two interleaved purposes: 1) it efficiently represents multiple versions of compiler plans, and 2) it guides the rewrite process. The layout of the e-graph guides which rewrite rules can be applied over its structure based on pattern matching as explained in Section 2.2. Given some rewrite rules are more beneficial than others, it is paramount to be able to quantify the significance of a node neighbourhood in the e-graph. At a micro level, it follows that different node neighbourhoods may be more important than others, warranting an attention-based method.

Graph Neural Networks (GNNs) [68] are ANNs for learning over graphs. The key idea behind GNNs is to use pairwise message passing between graph nodes to update their latent representations. The Graph Attention Network (GAT), introduced by Velickovic et al. [59], is an attention-based architecture for graph-structured data. A key advantage of GATs over other types of graph learning methods, such as Graph Convolutional Networks

(GCN) [23], is their ability to automatically infer the importance of graph substructures. This is achieved by injecting self-attention at the node level. This is formally expressed as part of succeeding work on GATs [5] as follows:

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j, \tag{3.1}$$

Equation 3.1 renders the graph attention convolution for a given node. The first term, $\alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i$ quantifies the contribution of the target node to its own latent representation. The second term represents the contribution of the neighbourhood to the representation of the target node. The attention coefficients are denoted $\alpha_{i,j}$ and they calibrate the impact of different nodes in the graph. In our proposed agent architecture, a 3-layer GAT is introduced to model a spatial representation of the encoded e-graph. As a preprocessing step, we inject the graph normalisation technique proposed by [6] to improve convergence.

Learning the Temporal Link

Recurrent neural networks (RNN) [33] are ANNs specialised for learning over sequential data. The key innovation behind RNNs is the ability to memorize previous steps to influence subsequent computations.

We leverage an RNN to model the trajectory in the e-graph construction MDP. The motivation is two-fold. First, the RNN is employed to capture the purely additive nature of the transition function given certain node neighbourhoods will not change throughout an episode. Second, we aim to mitigate some of the expressiveness pitfalls of the graph encoder. As an example, a global mean pool function may project two-distinct e-graphs into the same latent representation, leading to a sub-optimal policy.

To embed the temporal dynamics of the MDP, the system employs a Long Short-Term Memory (LSTM) [50, 19]. The motivation for using an LSTM is to prevent the vanishing gradient problem [18] over long construction episodes. The LSTM is formally defined below:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{3.2}$$

$$h_t = o_t \odot \tanh(c_t) \tag{3.3}$$

In Equation 3.2, c_t represents a memory cell. Intuitively, this collects information from the state space. There are two gates: an input gate denoted i_t and a forget gate denoted f_t. If the input gate is active, then the newly learnt representation g_t over the current state is stored. If the forget gate is on, then some information from the preceding cell c_{t-1} is discarded. Finally, the information from the cell is passed to the hidden state h_t, with a dependency on the output gate o_t. The flow of data through the LSTM layer is learnt during training via gradient updates.

Spatio-temporal PPO

The final ingredient in the deep model is represented by the *actor* and the *critic* networks (V and π in Figure 3.5). While previous work from the realm has employed a shared network with two output heads [39], our empirical analysis has shown that connecting two distinct linear layers, one for the actor and one for the critic, provides sufficient expressiveness. Furthermore, this keeps the size of the model relatively small. We can now describe the agent in a closed form as follows:

$$x_t = GAT(s_t) \tag{3.4}$$

$$z_t = GlobalPool(x_t) \tag{3.5}$$

$$h_t, c_t = LSTM(z_t, h_{t-1}, c_{t-1}) \tag{3.6}$$

$$\pi_{\theta_k} = f_{actor}(h_t) \tag{3.7}$$

$$V_{\theta_k} = f_{critic}(h_t) \tag{3.8}$$

In the equations above, θ_k represents the parameters of the entire deep model at iteration k. It is important to note that we train jointly the entire deep model by optimising the PPO-clip objective function rendered in Equation 2.3. We render the full workflow of the spatio-temporal policy with the pseudo-code below:

Algorithm 1 Spatio-temporal PPO

```
Require: Initialise model parameters $\theta_k$
1: for $k \leftarrow 1$ to $n$ do
2:     Collect a set of trajectories $D_k$ by rolling out $\pi_{\theta_k}$ in the environment
3:     Sample sequential data points from the set, as the order is important for LSTM
4:     Compute advantages $\hat{A}_t$ using $V_{\theta_k}$ (Equation 3.8)
5:     Compute policy loss with PPO-Clip
           $L^{CLIP}(\theta) = -\hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$
6:     Compute value function loss
           $L^{VF}(\theta) = \hat{\mathbb{E}}_t \left[(V_t - V_t^{\text{target}})^2\right]$
7:     Optimise jointly over the two losses by taking into account action entropy
8: end for
```

Algorithm 1 renders a high-level overview of the training routine. The routine extends PPO to accommodate temporal modelling. In contrast to PPO, which can perform gradient updates over randomly-indexed data points, we highlight that the training data in our case is sequential, as this is required to model the temporal link (line 3). The initial LSTM hidden states and memory cells are instantiated with values of zero and are reset at the end of an MDP episode. This implementation detail follows the best practices as addressed in previous literature [20].

3.4.3 Motivating Example

In this section, we discuss the advantage of modelling the temporal dynamics of equality saturation. It is paramount to ask ourselves whether a graph encoder is sufficient to devise a competitive policy on its own.

The basic intuition behind the architecture is that equality saturation is a sequential algorithm. At each timestep, the e-graph is expanded with compiler plans that meet the equivalence condition. Furthermore, large structures from the e-graph remain static during an expansion episode, as the algorithm is purely additive.

From a different angle, we take note that learning node representations over an e-graph is non-trivial. In practice, e-graphs may contain large cycles and in the pursuit of a graph-level prediction, we may lose information that distinguishes two e-graphs. We show how a GNN-based policy may fail by using a 4-state MDP.

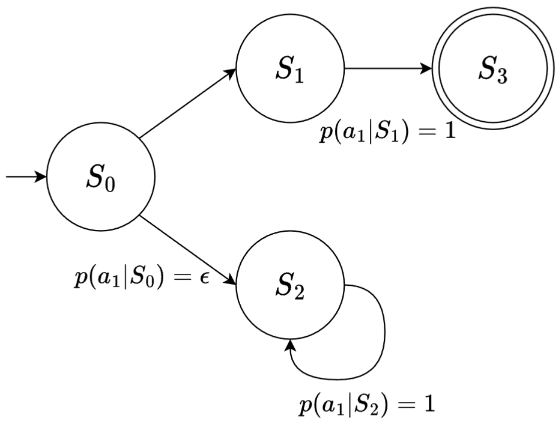


Figure 3.6: A 4-state finite MDP. The states are abstractions for e-graphs

Figure 3.6 renders a 4-state MDP over four e-graph states. State s_0 represents the initial SQL e-graph. We assume a sub-optimal policy is learnt that takes rewrite action a_1 in state s_0 with a probability $p(a_1|s_0) = \epsilon$. We further assume that action a_1 rewrites the e-graph s_1 into an optimal e-graph for extraction, s_3. If e-graphs s_1 and s_2 cannot be distinguished by the graph encoder (for instance by applying a global mean pool over two e-graphs of different sizes) and rewrite a_1 has already been saturated for s_2, then the *regret* is proportional to the horizon. In this context, we define regret as the number of times the wrong action has been taken by the agent during an episode of horizon T.

While we cannot generalise our findings to all types of graph learning architectures, we deem they provide enough incentive to research temporal policies when performing deep RL over graph states.

3.5 Query Cost & Extraction Procedure

The final instrument in Aurora is the e-graph extraction routine (step 5 in Figure 3.2). The aim of this routine is to extract the optimal plan from the query e-graph. It is important to remember the e-graph is an efficient encoding of multiple equivalent query plans. See Section 2.2 for equality saturation and the extraction procedure.

3.5.1 Query Cost Model

Extracting the optimal plan from the e-graph is non-trivial in the database scenario. While previous work from the realm mainly focused on the size of the expression (i.e., the number of nodes in the plan) as a cost function [52], we deem this provides no intuition regarding the performance of a query plan. To this end, it is paramount to devise a cost function over the SQL e-graph that can be linked back to the end-to-end latency of a query plan. In our implementation, we try variations of the cost model proposed by [45].

First, a proxy for the size of the data that flows through the query plan is defined. In this context, the size of the data is proportional to the intermediate result returned by an operator. For example, the intermediary output returned by a `SCAN` operation over a table T has a size of $num\_rows_T \times width_T$. If the `SCAN` operator is followed by a `FILTER` operation, which without loss of generality we assume reduces the number of tuples by 50%, then the new intermediary result will have a size $\frac{num\_rows_T}{2} \times width_T$. Based on the same reasoning, a projection that selects 50% of the columns will produce an output of size $num\_rows_T \times \frac{width_T}{2}$. As shown in Section 3.4.1, the size of the intermediary results can be stored in e-classes at runtime. This is because all operators (e-nodes) from an e-class should produce equivalent outputs.

Second, an operator-based cost function is implemented over the size of the intermediary result. Each e-node is either a plan operator, a column, or a table. The tables and columns are not assigned a cost, as they only facilitate execution by identifying relevant bodies from the database schema. Plan operators are assigned costs based on asymptotic bounds. For instance, if a table has n tuples and m columns, then a `SCAN` operation is assumed to cost $\mathcal{O}(nm)$.

3.5.2 Extraction with Integer Linear Programming

Having established the operator costs, an extraction scheme is then needed to select the cheapest operators from a query e-graph, operators that form an executable SQL plan. A naive scheme is to extract the operators in a greedy bottom-up fashion. An example is rendered in Figure 3.7.

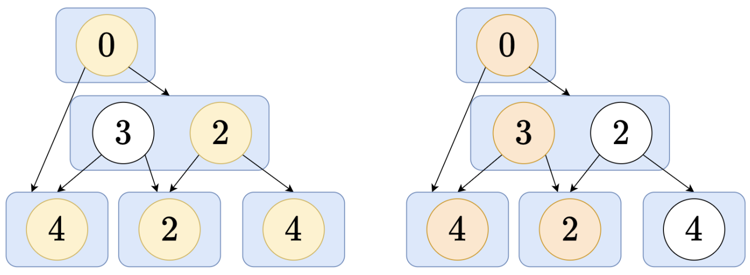


Figure 3.7: Extraction procedure over an e-graph. The e-nodes are marked with their costs.

Figure 3.7 shows the output of two extraction procedures over the same e-graph. We first look at the left extraction scheme, which follows a greedy bottom-up extractor. The e-node with cost 2 will be picked over the e-node with cost 3, thus leading to a sub-optimal execution plan. The right e-graph renders the optimal sequence of operators (depicted in orange), which yields a total cost of 9 instead of a greedy cost of 12. This follows from the assumption that an operator will not be executed twice if two e-nodes point to the same sub-tree.

We have shown a greedy bottom-up extraction is suboptimal. To address this, we formulate the extraction procedure as a constrained optimisation problem by using integer linear programming. Integer linear programming (ILP) proxies an optimisation problem through linear constraints over integer decision variables. This is a common approach in equality saturation literature [60, 65]. We solve the ILP with SCIP [1], a non-commercial solver.

Problem Formulation. Fix E the set of e-classes in the e-graph, where the root e-class is indexed by 0, and $\hat{E}_j$ the set of e-nodes in e-class j. Denote K the set of e-nodes in the e-graph. For each operator i in K, we introduce a Boolean decision variable y_i. Each e-class j in E is characterised by an integer decision variable z_j. Furthermore, we denote L_i the set of e-class children for e-node i and P_i the parent e-class for e-node i. Each operator has a specific cost as explained above. We denote the operator cost c. Then we extract the query plan from the e-graph according to:

$$\begin{aligned}
\text{Minimise:}\quad & \sum_i y_i \cdot c_i \\
\text{subject to:}\quad & y_i \in \{0,1\} \\
& 0 \leq z_j < |E|, z_j \in \mathbb{N} \\
& \sum_{i \in \hat{E}_0} y_i = 1 \quad \text{(Constraint 1)} \\
& \forall i \in K, \forall j \in L_i, y_i \leq \sum_{h \in \hat{E}_j} y_h \quad \text{(Constraint 2)} \\
& \forall i \in K, \forall j \in L_i, z_{P_i} - z_j + |E| \cdot (1 - y_i) \geq 1 \quad \text{(Constraint 3)}
\end{aligned} \tag{3.9}$$

The objective function is to minimise the cost of the SQL plan, by computing the total cost over the selected operators. Constraint 1 shows that only one operator needs to be picked from the root e-class. This follows from the fact that a SQL plan needs to be a tree. Constraint 2 describes the solver has to pick at least one node from each of an operator's children, which are e-classes by definition. We associate the e-graph with an AND-OR graph. The e-classes are OR nodes and the e-nodes are AND nodes. We ensure data flows upstream with no gaps by connecting all operator nodes until a leaf e-class is reached. Constraint 3 enforces a topological ordering over the e-classes in the extracted plan. A topological ordering is required to extract an executable directed acyclic graph (DAG), as the original e-graph may include cycles.

In practice, we alternate between a greedy bottom-up extraction and the ILP extractor for efficiency. For instance, during RL training we leverage a greedy approximation to compute the reward (recall the reward is computed based on the most competitive plan encoded in the e-graph), as the greedy extractor is orders of magnitude faster than the ILP solver. If the learnt policy is used for inference, the system can leverage the ILP solver to extract the optimal plan.

Chapter 4

Evaluation

The purpose of this chapter is to analyse the performance of the proposed system, Aurora. The hypotheses presented in Section 3.3 are attacked from two angles. First, the performance of the query rewrite mechanism is evaluated on a complex suite of analytical SQL queries. Second, the performance of the proposed RL algorithm is evaluated against egg, the state-of-the-art for equality saturation, Omelette, and a heuristic agent, across a number of tasks. Concretely, the overarching aim of this chapter is to unearth that:

1. Aurora can rewrite complex database queries to *significantly* reduce the end-to-end latency, thus addressing Hypothesis 1

2. Aurora is generalisable across different database management systems, thus addressing Hypothesis 2.

3. Aurora is able to find lower cost solutions than the baselines for equality saturation including Omelette, thus addressing Hypothesis 3.

4. The RL algorithm behind Aurora provides unique advantages over Omelette and these advantages can be generalised across other languages than relational algebra, thus proving Hypothesis 4.

This chapter is organized into three overarching sections. The first section delves into the technical details of the experimental setup. This delivers information about the RL agent's configuration and discusses the baselines as well as the test DBMSs. The subsequent section renders experimental results linked to the behaviour of the RL agent. The final section compares Aurora's RL algorithm with Omelette. The discussion is centred on *why* Aurora is able to surpass Omelette in the database domain and in mathematical simplification.

4.1 Experimental Setup

The empirical analysis is focused on evaluating the ability of Aurora to optimise database queries, as well as other types of expressions. We use the word *expression* because the intrinsic value of the proposed methodology is not restricted to the database domain, and in fact, it applies to any compiler language with limited changes. Concretely, Aurora can be directly applied to simplify mathematical expressions, and we unearth its effectiveness in this compiler domain in a study case in Section 4.4.1.

4.1.1 Dataset

It is paramount that Aurora is able to optimise a wide range of SQL queries. Our proposed equality saturation solver has to be effective at rewriting complex database transactions, such as multi-join analytical queries. To achieve the desired level of rewrite difficulty, we experiment with the TPC-H database benchmark [41]. The TPC-H benchmark is an online analytical processing (OLAP) workload. The database schema contains 62 columns over eight tables. The queries simulate decision support transactions associated with warehousing environments. We extract a subset of queries from the benchmark for our experimental analysis, based on the availability of operators in Risinglight.

4.1.2 Testbeds

There are two database management systems (DBMSs) we leverage in our experimental analysis. To benefit from a smooth integration between egg and the relational algebra language, we build Aurora on top of Risinglight. We remark it cardinal to show the efficiency of the query rewrite pipeline in mainstream databases. Henceforth, we turn to PostgreSQL for this purpose.

1. **Risinglight** [45] is a DBMS implemented in Rust. It exhibits a database query engine that independently confines rewrite rules. Aurora expands the set of rewrite rules from Risinglight and extends the engine to support user-defined rewrite orders. The plans devised by Aurora can be directly executed against the storage system without converting the plan back into a SQL dialect.

2. **PostgreSQL** [42] is a popular mainstream DBMS with wide applications in both industry as well as research. To execute queries on PostgreSQL, the query plans devised by Aurora are translated back into the Postgres dialect.

4.1.3 Baselines

The performance of equality saturation in the query rewrite domain is evaluated through several metrics (Table 4.1).

Metric	Explanation
Estimated cost	The estimated cost of a plan serves as a proxy for the true latency. This is implemented as shown in Section 3.5
Planning latency	The planning latency represents the sum between the cost of constructing the e-graph (cost of rewrite) and the extraction latency
Execution latency	The execution latency is the latency associated with fetching the data from the storage system
Total latency	planning latency + execution latency

Table 4.1: The core metrics used during evaluation. Other metrics such as the number of e-graph nodes are discussed.

The algorithms used to perform equality saturation serve as the baselines. Essentially, we quantify the extent to which each rewrite algorithm improves the defined metrics.

1. **Egg** [64] is the state-of-the art for equality saturation. It follows a sequential application of rewrite rules to build the e-graph until saturation is achieved or a node limit is reached.

2. **Omelette** [53, 52] is an RL-based method for guiding the application of rewrite rules in equality saturation. It features an actor-critic architecture with a 3-layer Graph Attention Network policy.

3. **Heuristic** is a pseudo-random agent we propose to test the hypothesis that RL is competitive. The agent has control over the same range of actions as the RL policy. The motivation is that random search is surprisingly competitive to RL across a wide range of domains [29].

4.1.4 Agent Configuration

For each optimisation task, we train two distinct agents, one with Aurora, which uses the spatio-temporal RL defined in Section 3.4, and one with Omelette which follows a classic PPO training routine. Given the complexity associated with training two RL agents, and the notoriously large space of hyperparameters PPO exhibits, we interleave reputable settings from off-the-shelf RL [21] with hand-tuning to set hyperparameters (see A.1 for reproducibility).

The training routine for both algorithms is task-oriented. For each combinatorial optimisation task, the agents are trained over 200k steps where each episode has a horizon of 200 steps. This is a sufficient budget for most queries to be rewritten in a form that significantly improves upon the original. In some cases, applying 200 rewrite rules is rather wasteful, so the agent learns to stop the optimisation session early. This is incentivised by the reward function as it penalises the additional overhead to the planning latency. In more complex queries, such as the ones exhibiting multi-joins, an e-graph node limit is usually reached before the agent chooses to stop on its own. Unless otherwise noted, all the experimental results are averaged over 5 seeds. The same seeds are used for all algorithms. For any given seed, given the trained policies are stochastic, the pre-trained agents are rolled out in the environment for 100 roll-outs, collecting only the best results. The same applies to the heuristic agent. This provides reasonable statistical significance across the results. Equality saturation with egg is run only once given it is deterministic.

The system used for training exhibits a Tesla P100 GPU with 16GB of VRAM, 24 logical CPUs, and 64GB of memory.

4.2 Why RL for DB-oriented Equality Saturation?

We start our empirical evaluation by analysing how to rewrite query 5 from the TPC-H benchmark (Listing 4.1) with equality saturation. The purpose of this is twofold: 1) to explain the moving parts of equality saturation for query rewrite and 2) to give empirical motivation for introducing reinforcement learning within the wider framework.

Listing 4.1: TPC-H Query 5

```
SELECT
    n_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM
    customer, orders, lineitem, supplier, nation, region
WHERE
    c_custkey = o_custkey AND l_orderkey = o_orderkey
    AND l_suppkey = s_suppkey AND c_nationkey = s_nationkey
    AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND
       r_name = 'AFRICA' AND o_orderdate >= DATE '1994-01-01'
    AND o_orderdate < DATE '1994-01-01' + INTERVAL '1' YEAR
GROUP BY
    n_name
ORDER BY
    revenue DESC;
```

Query 5 is a 6-table join analytical query characterised by a diverse range of operators ranging from `SUM` to `ORDER BY`. This makes it a good case study for the query rewrite problem. In the first experiment, we use the egg library and the previously-introduced rewrite rules (see Table 3.2 for the rewrite rule clusters) to optimise the query over our cost model (Section 3.5).

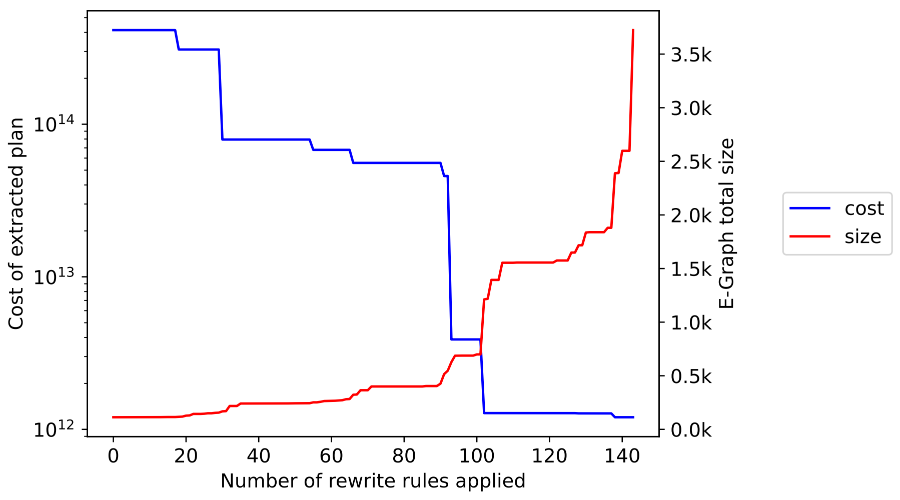


Figure 4.1: Results for Query 5. The relation between the size of the e-graph and the estimated cost of the extracted SQL plan. The number of nodes is limited to 3.5k. **Red line:** the growth trend for the number of nodes in the e-graph. **Blue line:** the cost of the extracted plan after each rewrite rule.

Figure 4.1 shows the link between the size of the e-graph and the cost of the extracted plan after each e-graph expansion. As expected, as more equivalent SQL expressions are added to the e-graph, the cost of the extracted SQL plan decreases, while the number of nodes in the e-graph increases. Intuitively, when a rewrite rule is applied over the e-graph, new e-nodes and e-classes are created to represent the alternative SQL plans. The process is non-destructive, and the original plan remains encoded in the e-graph.

A clear observation is that the node increase is not proportional to the cost improvement. For instance, we remark how the size of the e-graph explodes after the first 140 rule applications, while the cost remains rather constant during the same window. Given egg applies the rewrite rules sequentially, this raises the question of whether a plan with a lower cost can be found within the same node limit, by changing the order in which the rules are applied as well as their frequency.

Figure 4.2 demonstrates that repurposing the classic equality saturation algorithm (Listing 2.1) for the query rewrite problem is suboptimal, with two caveats: 1) the e-graph node limit is a concern and therefore bounded (e.g. egg is set to stop upon reaching 10k nodes by default) and 2) the e-graph does not get saturated within the node limit. Upon reaching saturation, the e-graph encodes all possible plans so the extracted plans will have the same cost (globally optimal). As we have established in Section 3.2, saturation is not a feasible goal in online systems due to the stringent memory requirements. Henceforth, a wise application of rules, like the ones learnt by Aurora, can minimise the computation overhead while producing competitive plans. For comparison purposes, we render the extracted plans in Table 4.2.

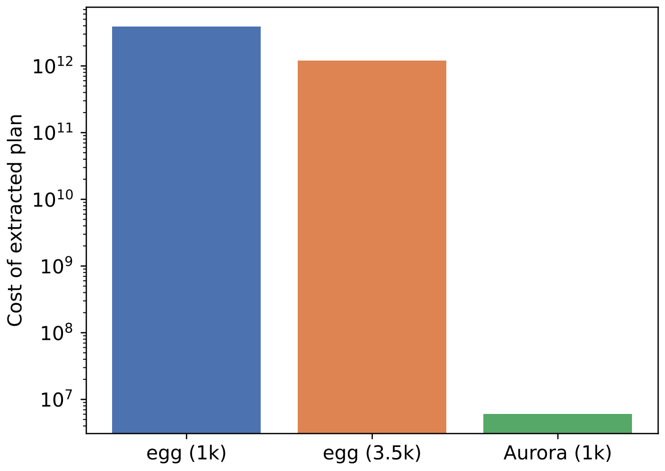


Figure 4.2: Results for Query 5. The costs of the extracted plans from three different e-graphs. The first two e-graphs are grown by a sequential application of rewrite rules up to 1.5k and 3.5k e-graph nodes, respectively. The right-most plan is extracted from an e-graph of 1k nodes expanded with a learnt rewrite order. Lower cost is better.

Extracted plan with egg (3.5k)

```
Projection
├── exprs:
│   ┌── $0.1
│   └── sum
│       └── * { lhs: $7.5, rhs: - { lhs: 1, rhs: $7.6 } }
└── Order
    ├── by: desc
    │   └── sum
    │       └── * { lhs: $7.5, rhs: - { lhs: 1, rhs: $7.6 } }
    └── Aggregate
        ├── aggs: sum
        │   └── * { lhs: $7.5, rhs: - { lhs: 1, rhs: $7.6 } }
        ├── group_by: [ $0.1 ]
        └── Projection { exprs: [ $0.1, $7.5, $7.6 ] }
            └── Join
                ├── type: inner
                ├── on: and
                │   ├── lhs: = { lhs: $0.0, rhs: $3.3 }
                │   └── rhs: and
                │       ├── lhs: = { lhs: $3.3, rhs: $5.3 }
                │       └── rhs: and
                │           ├── lhs: = { lhs: $3.0, rhs: $7.2 }
                │           └── rhs: and { lhs: = { lhs: $7.0, rhs: $6.0 }, rhs: = { lhs: $6.1, rhs: $5.0 } }
                ├── Projection { exprs: [ $5.0, $5.3, $6.0, $6.1 ] }
                │   └── Join { type: inner }
                │       ├── Scan { table: $5, list: [ $5.0, $5.3 ] }
                │       └── Projection { exprs: [ $6.0, $6.1 ] }
                │           └── Filter
                │               ├── cond: and
                │               │   ├── lhs: > { lhs: 1995-01-01, rhs: $6.4 }
                │               │   └── rhs: >= { lhs: $6.4, rhs: 1994-01-01 }
                │               └── Scan { table: $6, list: [ $6.0, $6.1, $6.2, $6.3, $6.4, $6.5, $6.6, $6.7, $6.8 ]
                └── Projection { exprs: [ $0.0, $0.1, $3.0, $3.3, $7.0, $7.2, $7.5, $7.6 ] }
                    └── Join { type: inner }
                        ├── Scan { table: $7, list: [ $7.0, $7.2, $7.5, $7.6 ] }
                        └── Projection { exprs: [ $0.0, $0.1, $3.0, $3.3 ] }
                            └── Join { type: inner }
                                ├── Scan { table: $3, list: [ $3.0, $3.3 ] }
                                └── Projection { exprs: [ $0.0, $0.1 ] }
                                    └── Join
                                        ├── type: inner
                                        ├── on: and
                                        │   ├── lhs: = { lhs: $1.1, rhs: 'AFRICA' }
                                        │   └── rhs: = { lhs: $1.0, rhs: $0.2 }
                                        ├── Scan { table: $0, list: [ $0.0, $0.1, $0.2 ] }
                                        └── Scan { table: $1, list: [ $1.0, $1.1 ] }
```

Extracted plan with Aurora (1k)

```
Projection
├── exprs:
│   ┌── $0.1
│   └── sum
│       └── * { lhs: $7.5, rhs: - { lhs: 1, rhs: $7.6 } }
└── Order
    ├── by: desc
    │   └── sum
    │       └── * { lhs: $7.5, rhs: - { lhs: 1, rhs: $7.6 } }
    └── Aggregate
        ├── aggs: sum
        │   └── * { lhs: $7.5, rhs: - { lhs: 1, rhs: $7.6 } }
        ├── group_by: [ $0.1 ]
        └── Projection { exprs: [ $0.1, $7.5, $7.6 ] }
            └── Join { type: inner, on: = { lhs: $0.0, rhs: $3.3 } }
                ├── Projection { exprs: [ $3.3, $7.5, $7.6 ] }
                │   └── HashJoin { type: inner, on: = { lhs: [ $5.0, $5.3 ], rhs: [ $6.1, $3.3 ] } }
                │       ├── Scan { table: $5, list: [ $5.0, $5.3 ] }
                │       └── Projection { exprs: [ $3.3, $6.1, $7.5, $7.6 ] }
                │           └── HashJoin { type: inner, on: = { lhs: [ $6.0 ], rhs: [ $7.0 ] } }
                │               ├── Projection { exprs: [ $6.0, $6.1 ] }
                │               │   └── Filter
                │               │       ├── cond: and
                │               │       │   ├── lhs: >= { lhs: $6.4, rhs: 1994-01-01 }
                │               │       │   └── rhs: > { lhs: 1995-01-01, rhs: $6.4 }
                │               │       └── Projection { exprs: [ $6.0, $6.1, $6.4 ] }
                │               │           └── Scan { table: $6, list: [ $6.0, $6.1, $6.4 ] }
                │               └── Projection { exprs: [ $3.3, $7.0, $7.5, $7.6 ] }
                │                   └── HashJoin { type: inner, on: = { lhs: [ $7.2 ], rhs: [ $3.0 ] } }
                │                       ├── Scan { table: $7, list: [ $7.0, $7.2, $7.5, $7.6 ] }
                │                       └── Scan { table: $3, list: [ $3.0, $3.3 ] }
                └── Projection { exprs: [ $0.0, $0.1 ] }
                    └── Join { type: inner, on: = { lhs: $1.0, rhs: $0.2 } }
                        ├── Scan { table: $0, list: [ $0.0, $0.1, $0.2 ] }
                        └── Projection { exprs: [ $1.0 ] }
                            └── Filter { cond: = { lhs: $1.1, rhs: 'AFRICA' } }
                                └── Scan { table: $1, list: [ $1.0, $1.1, $1.2 ] }
```

Table 4.2: Alternative plans for Query 5. Extracted by growing e-graphs with egg (3.5k node limit) and Aurora (1k node limit). The tables are encoded as \$0\$, \$1\$, etc. The columns for table \$0\$ are encoded as \$0.0, \$0.1\$ etc. The schema is constant across the two plans

Table 4.2 displays the plans associated with the cost results from Figure 4.2. Without delving into the underpinning intricacies, we remark that the plan on the left has suboptimal blocks. For instance, a `SCAN` operation is performed over all the columns in table \$6\$. In turn, the plan devised by Aurora does not repeat the same mistake, showing the algorithm has learnt to push the projection as early in the plan as possible. Furthermore, we observe the introduction of `HASH-JOIN` operators in this plan, which are generally more efficient.

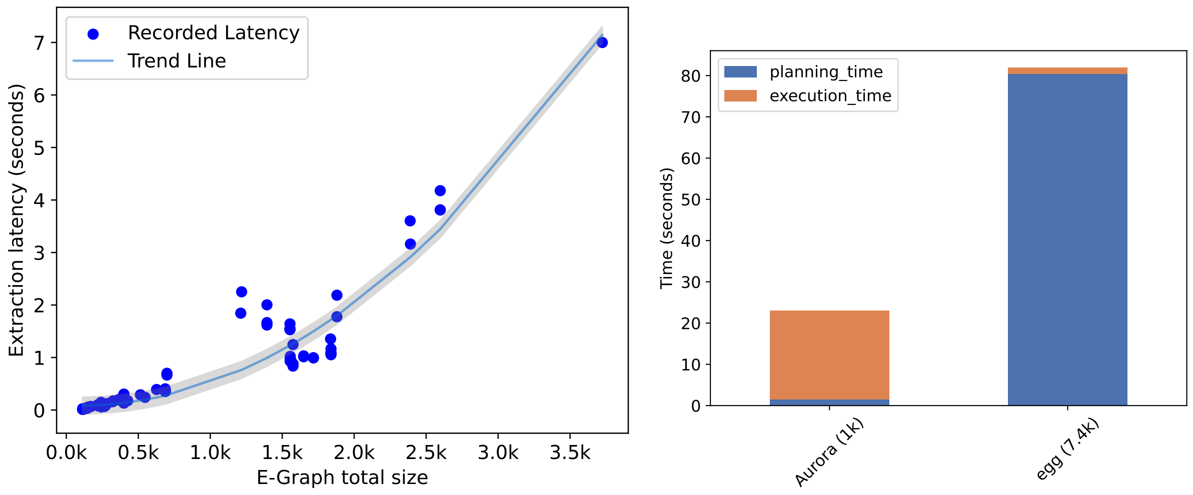


Figure 4.3: **Left:** The relation between the size of the e-graph and the extraction latency. **Right:** Planning and execution statistics for two alternative plans executed on Risinglight. To extract the SQL plan we solve the ILP described in Section 3.5.

Figure 4.3 renders insights linked to the overhead of large e-graph structures through query planning latency. Concretely, in the left plot, we observe the connection between the latency incurred for extracting the best SQL plan and the size of the underpinning e-graph. As the number of equivalent plans grows, extracting the optimal SQL plan from the e-graph incurs a higher cost. In the right plot, we measure the planning time (including extraction), as well as the execution time for two Query 5 plans devised with flavours of equality saturation. The first plan is the one devised by Aurora in Table 4.2. The execution time is approximately 20 seconds with a standard deviation of $2s$, and a planning latency in the order of milliseconds. The second plan is extracted from an e-graph of $7.3k$ nodes built using egg. In this case, the planning latency has a mean of 78 seconds, with a large percentage associated with extracting the optimal SQL plan from the e-graph. The execution time is discarded as the planning time is already over 3 times higher than the total latency for the first plan. This demonstrates another reason to restrict the size of the e-graph in query optimisation, given the planning overhead has to be proportional to the execution speed-up.

4.3 Performance Results

This section investigates whether Aurora brings the hypothesized benefits from a performance perspective. The driver for the optimisation process is the estimated cost of a plan. This serves as a proxy for the true latency without the additional overhead of actually executing a query. The combinatorial problem is constrained by an e-graph node limit of 1000 nodes. This serves two purposes: 1) to account for the planning latency associated with higher-node e-graphs, and 2) to isolate the performance of the RL algorithms within the equality saturation framework.

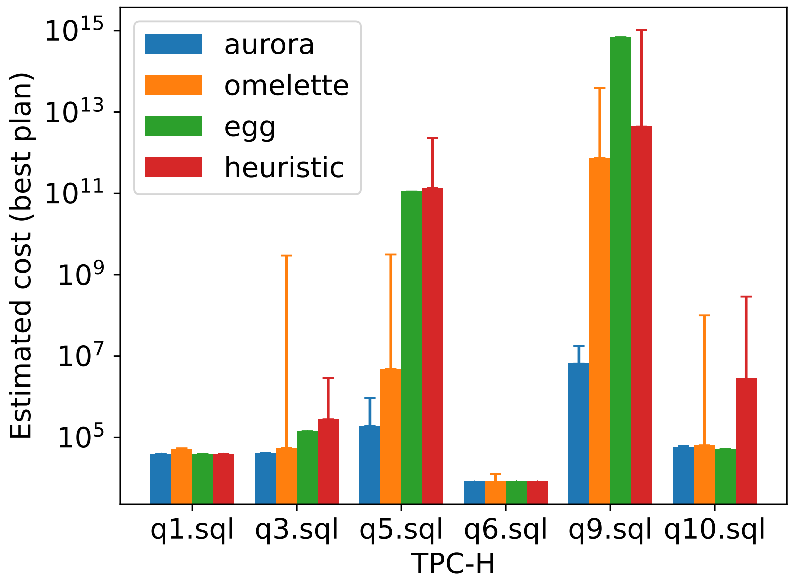


Figure 4.4: The median cost of the extracted plan by algorithm over 5 seeds. The whiskers represent the maximums. Lower cost is better.

Figure 4.4 summarises the performance across the evaluated algorithms. The metric we evaluate is the estimated cost of the extracted plan (see Section 3.5). It is evident from the plot that Aurora discovers SQL plans that are orders of magnitude lower in cost than the baselines. The compelling results provide a stepping stone to answering *Hypothesis 3*, which questions whether Aurora can find lower-cost solutions than other equality saturation algorithms.

In contrast to previous findings [52], we observe that a random application of rules yields higher-cost plans than a sequential application of rules, as per egg. The significance of this result is two-fold. First, it shows that some structure is needed to solve the SQL e-graph construction problem. While the original author of Omelette remarks random search improves upon egg by 20% cost-wise, we associate this to the inherent simplicity of the domains used, constituted of simplifying first-order logic and mathematical expressions. In the original work, each domain has below 27 possible rewrite actions, making it feasible for the random agent to better *guess* a competitive order. However, when faced with the SQL domain, the lack of structure in random search plays a detrimental role. Second, it warrants that RL is a competitive choice for optimising SQL plans over the space spanned by e-graphs. This follows from the performance gap between the RL agents and the heuristic search.

Of particular importance, we remark the steep improvements in TPC-H Query 5 and Query 9. There is notable complexity associated with Query 5 and Query 9 as they exhibit join operations over six tables each. This exacerbates the need for a wise application of rules. First, priority has to be given to rewrite rules that reduce the size of the data that flows through the plan. As a concrete example, a join operation between all the rows across two tables can lead to plans orders of magnitude more expensive than plans that join *filtered* tables. A filtered table is a table that contains a subset of the original tuples. Second, the rewrite rules have to expand the e-graph such that multiple join orders are possible. This aims to tackle the join ordering problem [54], an ever-present topic in database literature [22, 30, 31].

In all cases, the RL agents tend to outperform the other baselines. However, Aurora exhibits much lower variance than Omelette, while also providing improvement of up to 99% over Omelette in some instances.

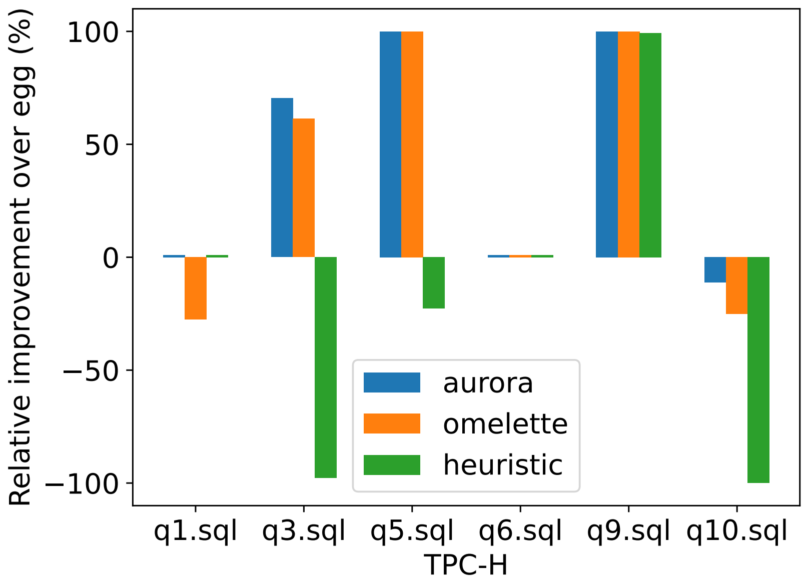
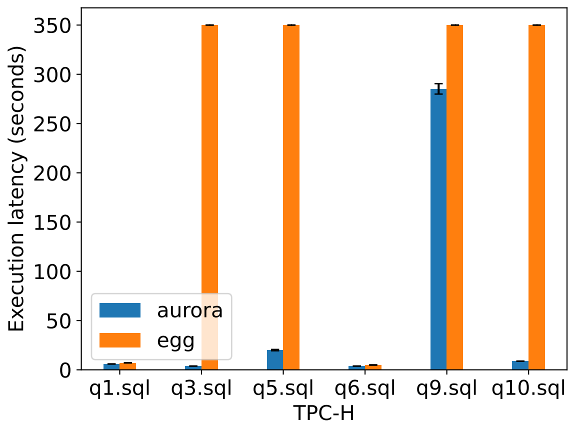


Figure 4.5: The relative improvement over **egg**, the state-of-the-art equality saturation solver. The negative outcome is bounded at -100%. Higher relative improvement is better.

Figure 4.5 quantifies the relative improvement of each algorithm over egg. On one hand, from the plots, we draw the conclusion that RL methods are competitive at guiding the application of rules in equality saturation, with a relative improvement of 99% over the sequential method for 33% of the queries. On the other hand, we remark that Omelette performs worse than a sequential application of rules in two cases. We provide an in-depth analysis of this behaviour in Section 4.4.

It is important to note that while the relative improvement is reasonably similar between algorithms in some cases (Query 9), the final costs can be wildly different due to the magnitude of estimated costs we operate on (see Figure 4.5).



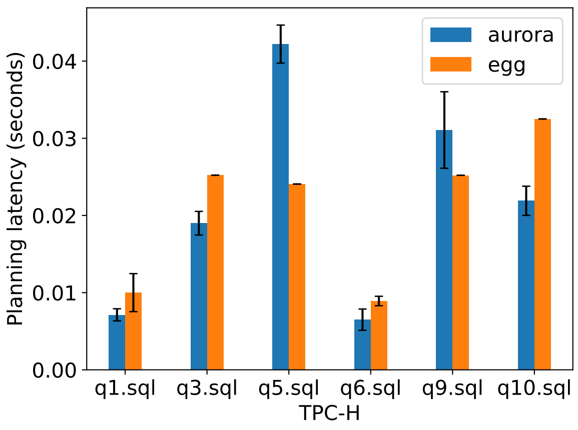


Figure 4.6: **Left:** the execution latency of the extracted SQL plan with a timeout set at 350 seconds. **Right:** the planning latency. All metrics are averaged over 5 trials. In both cases, lower is better.

Figure 4.6 summarises the query latency results. This is by far the most important metric in the SQL domain. While previous work in the realm has mostly focused on the offline application of equality saturation with costs available at runtime [65, 52], the SQL domain is different in this regard. First, during the optimisation routine, as well as during RL training, only a proxy of the real cost can be used. Extracting the real optimal plan from an e-graph would essentially require the real cost associated with the operators and naturally a perfect match of the ILP formulation to the true latency (Section 3.5). Estimating the cost of a query is an open problem in the database community, and classic optimisers usually leverage statistics from the database catalogue to pick a plan [32, 55]. Existing research focuses on learning the real cost across individual SQL plans, however, the application of such methods over e-graphs establishes a promising future work avenue.

The left plot from Figure 4.6 renders the execution latency. The execution DBMS is Risinglight. We remark that a guided application of rules within equality saturation, such as the one learnt by Aurora, greatly improves upon the sequential strategy. We set a timeout of 350 seconds for the execution to finish given the rather small size of the data pool (4.7GB). In 4 out of 6 cases, egg runs into a timeout. This is consistent across the 5 trials we test over. The importance of this result is two-fold. First, it is clear that Aurora can find non-trivial SQL plans when compared to a simple sequential application of rewrite rules. Second, the cost model we define is generally a good proxy for the true latency, otherwise, the cost function would have not been correlated with it across the diverse range of queries we evaluate on. This result provides a partial answer to *Hypothesis 1*, which questions whether Aurora can rewrite SQL plans into significantly better ones.

The planning latency is rendered in the right plot. A core motivating factor for introducing RL in equality saturation is to mitigate the overhead associated with large e-graph structures while obtaining competitive SQL plans within the node constraints. The planning overhead associated with Aurora is higher for Query 5 and Query 9. This is expected, as a wise e-graph construction routine should allow for more rewrites while keeping the e-graph within the node limit. Furthermore, the *Reset* action described in Section 3.4.1 gives an incentive to the agent to try different construction routes. However, we remark the planning latency operates on an insignificant scale (milliseconds), which combined with our previous result on execution latency paints a compelling picture to prove *Hypothesis 1*, that Aurora indeed minimises the end-to-end query latency.

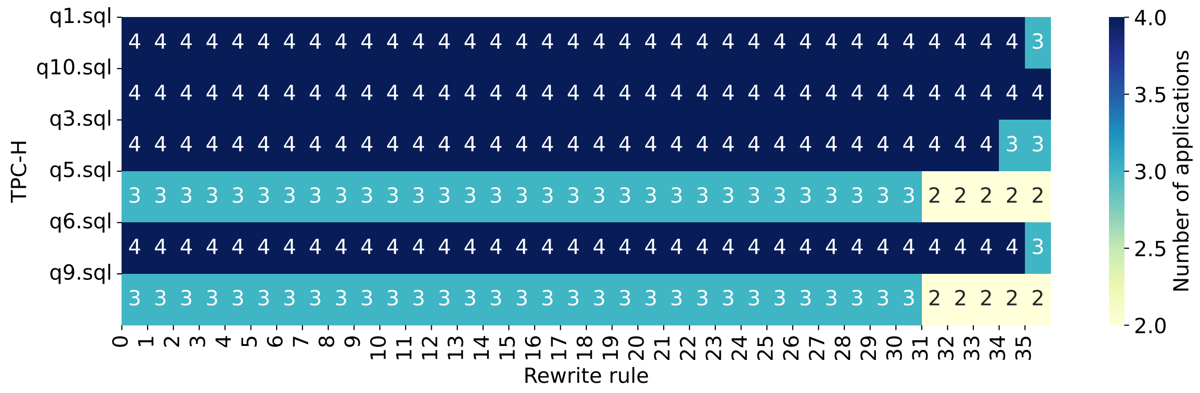


Figure 4.7: The rewrite rules applied by egg. The x-axis represents the index of the rewrite rule

To provide some intuition on why the learnt e-graph construction is advantageous, we show how egg operates in Figure 4.7. It is evident from the plot that egg tries to saturate the e-graph with a sequential application of rules. Figure 4.8 renders the rules applied by a pre-trained Aurora.

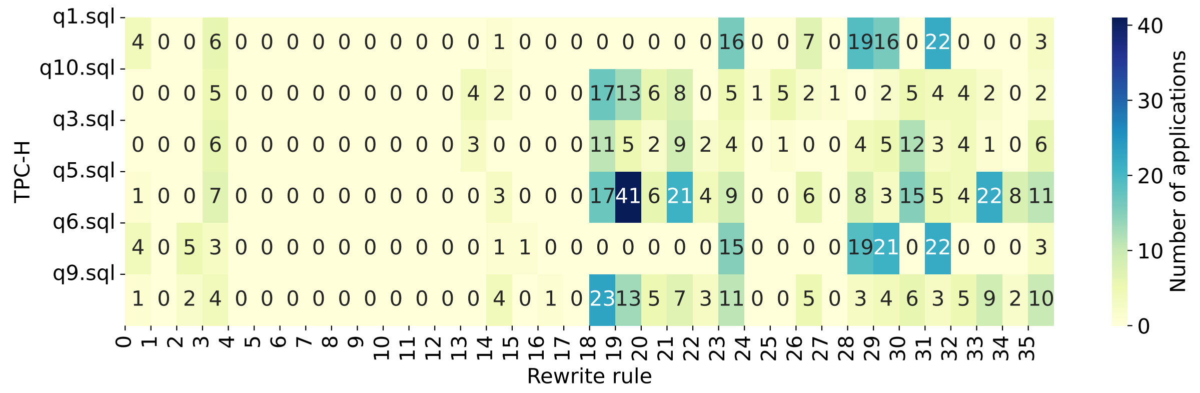


Figure 4.8: The actions chosen by pre-trained Aurora

Figure 4.8 displays the rewrite rules selected by a pre-trained Aurora policy during rollout. It is clear from the plot that the strategy is more focused than in egg. For instance, we identify a subset of rules (range R5-R13) that are never picked from the available suite, while other rules such as R18 and R19 are applied extensively in the multi-join queries. It is important to note that Query 1 and Query 6 do not have join operators. To get a clear view of the agent's behaviour, we list the most frequently selected rules in Table 4.3

Rule	**Explanation**
R19 pushdown-filter-join-left	pushes the filter predicate inside a join
R23 identical-proj	removes duplicate projections
R28 pushdown-proj-filter	puts the projection before a filter
R29 pushdown-proj-agg	puts the projection before an aggregation
R30 pushdown-proj-join	puts the projection before a join
R31 pushdown-proj-scan	puts the projection above a scan operator
R32 join-reorder	reorder the terms of a join operation

Table 4.3: Prominent rewrite rules picked by Aurora.

4.3.1 Case Study: PostgreSQL

This section presents a study case on PostgreSQL, a mainstream database system. The motivation behind the experiment is to show the techniques are DBMS-agnostic, to prove *Hypothesis 2*. We use Aurora to rewrite the *Alternative* query from Table 4.4 into the *Optimal* form. Subsequently, the rewritten query is executed in PostgreSQL.

Alias	**Query**
Original	*select o_orderkey from orders where o_orderstatus = 'F' and o_orderkey in (select o_orderkey from orders where o_totalprice > 40000)*
Alternative	*select key from (select o_orderkey as key, o_orderstatus as status, o_totalprice as price from orders where o_totalprice > 40000) as inter where status = 'F' and price > 40000*
Optimal	*select o_orderkey from orders where o_orderstatus = 'F' and o_totalprice > 40000*

Table 4.4: Equivalent queries with different levels of performance.

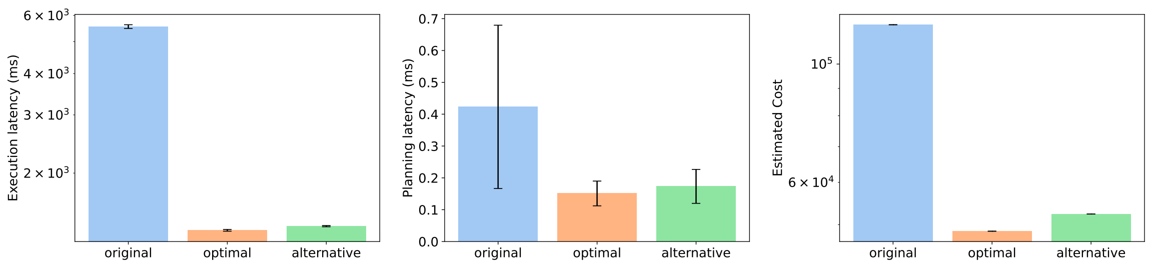


Figure 4.9: From left to right: the execution latency, the planning latency, and the estimated cost (PostgreSQL) of the queries from Table 4.4 (5 trials). Lower is better in all cases.

Figure 4.9 shows that Aurora can be generalised across database systems. We remark the first and second queries from Table 4.4 incur higher latency in PostgreSQL than the one devised by Aurora. In the experiment, Aurora receives the second query, coined *Alternative*, and returns the *Optimal*. The rewritten query is 11% faster. We observe the PostgreSQL optimiser does not take advantage of the correlation between the `FILTER` operation in the outer query and the one in the nested query when executing the former (i.e., *"price > 40000"*), resulting in higher latency (the plans are rendered in Appendix A.2). The *Original* query is added for reference to show Aurora can theoretically bring large improvements over mainstream DBs. However, Aurora currently does not support **IN** operators over nested queries, while PostgreSQL does. This engineering extension is left for future work.

4.4 Algorithm performance

This section analyses the behaviour of the spatio-temporal RL agent from Aurora. The baseline for comparison is Omelette. We first evaluate the agents in the database domain, and then evaluate their performance on simplifying mathematical expressions (see Table 3.2 for an example of a mathematical rule). The reason for the latter analysis is to show both methods are domain agnostic.

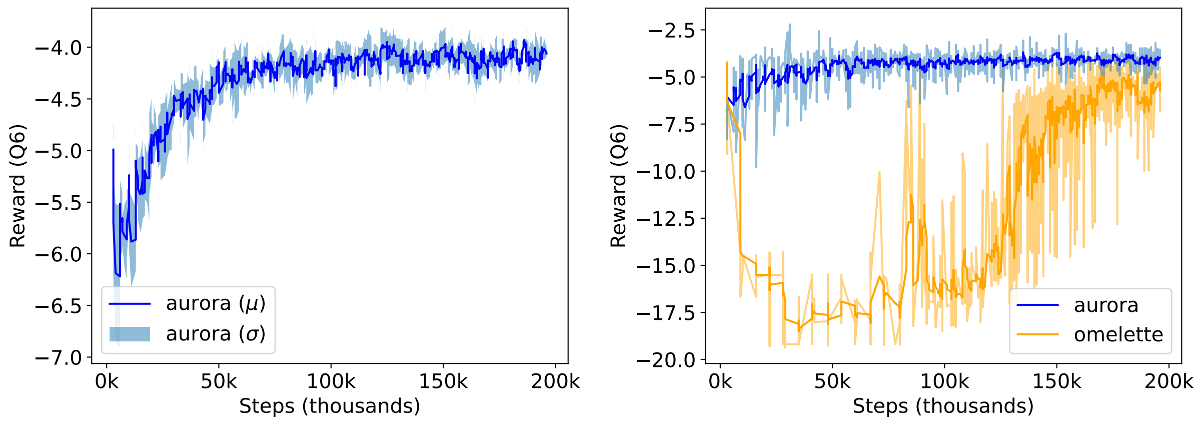


Figure 4.10: **Left:** Aurora convergence (fine-tune) curve on Query 6 from the TPC-H benchmark. **Right:** A comparison between Omelette and Aurora on optimising Query 6 (same seed). Higher reward is better.

Figure 4.10 renders insightful results linked to how Aurora and Omelette behave during training. We first look at the left plot to observe the mean and the standard deviation for Aurora. The reward curve is subject to an exponential smoothing routine and subsequently averaged across 5 seeds. The significance of this first result is to show that Aurora is able to learn to perform query rewrite over e-graphs. The negative scale of the reward is due to action masking (the agent is prevented to end the episode earlier than 100 rewrites), which is reasonably wasteful for Query 6. Nonetheless, the planning overhead associated with this is negligible.

The plot on the right shows a comparison between Omelette and Aurora on the same seed. While the initial divergence in Omelette cannot be justified over a single run, we observe similar patterns occur over multiple seeds, and across all the queries we have trained over.

We hypothesize the temporal representation learnt by Aurora improves the stability of the algorithm in comparison to Omelette which solely relies on learning a representation over the e-graph. Intuitively, by modelling the e-graph construction trajectory through an LSTM, the policy is less likely to be confused between similar e-graphs. In contrast, Omelette only leverages two 3-layer Graph Attention Networks for the value function and the policy. This raises the question of whether the GATs alone are able to distinguish between the deluge of e-graphs spanned by the SQL operators.

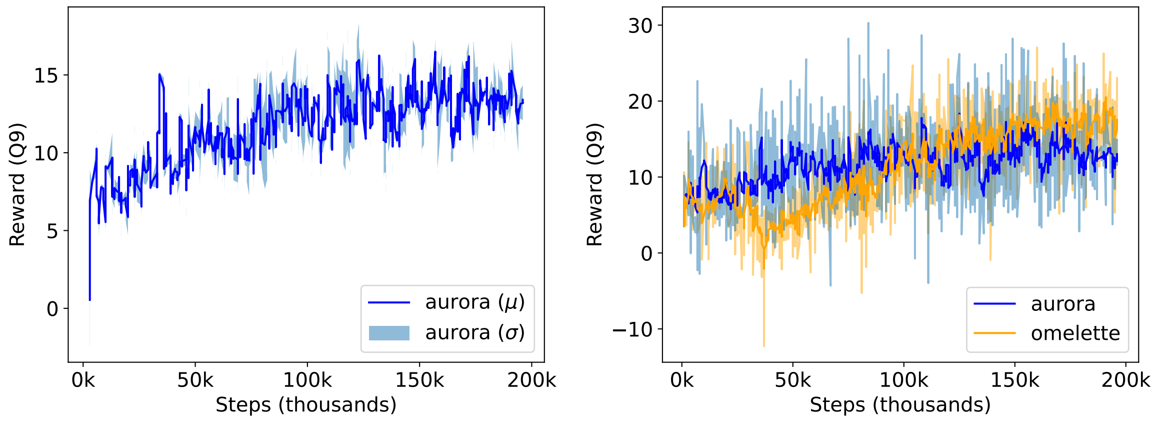


Figure 4.11: **Left:** Aurora convergence (fine-tune) curve on Query 9. **Right:** A comparison between Omelette and Aurora on optimising Query 9 (same seed). Higher reward is better.

Figure 4.11 renders the performance of the agents on Query 9 from the TPC-H benchmark. The right plot shows that Omelette, while it suffers from higher variance, it eventually finds a better e-graph construction path than Aurora. However, as we have observed in 4.4, it produces SQL plans of higher cost than Aurora on average.

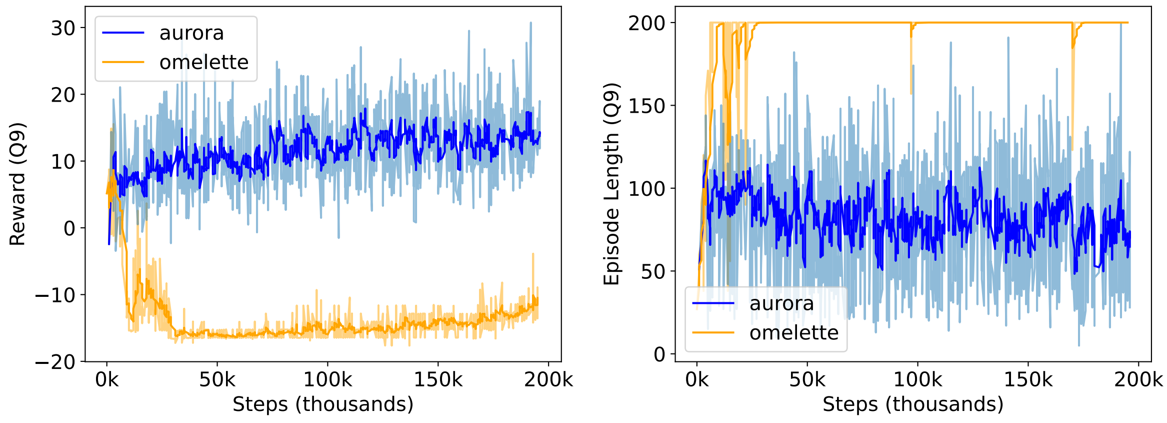


Figure 4.12: From left to right: The divergence of Omelette. The episodic length of Aurora and Omelette over the same seed. Higher reward is better. Lower episodic length is better.

Figure 4.12 shows the duality of Omelette. While the algorithm has the potential to find competitive SQL plans over the e-graph space, it is not uncommon to fail to converge.

Once again, we link the meagre performance of Omelette in the database domain to the complexity of the environment. In turn, Aurora is more stable across different seeds and queries. The trade-off is the higher computation requirements associated with the neural architecture, as it exhibits more parameters.

The divergence of Omelette in optimising Query 9 can be explained through the lens of the episode length. The plot on the right (Figure 4.12) provides evidence for Omelette's meagre learning curve. The policy enters 200-step episodes, despite our designed reward which penalises wasteful application of rules. We deem the stability of Aurora and the higher sample efficiency it exhibits in comparison to Omelette, to provide reasonable grounds to answer *Hypothesis 4*, which questions whether Aurora can bring an advantage over Omelette.

4.4.1 Case Study: Mathematical Simplification

For completeness, we also evaluate the two methods over mathematical simplification problems. This serves two purposes: 1) to prove Aurora is generalisable across different languages with little change, and 2) to complete our analysis of the two learning methods across multiple domains

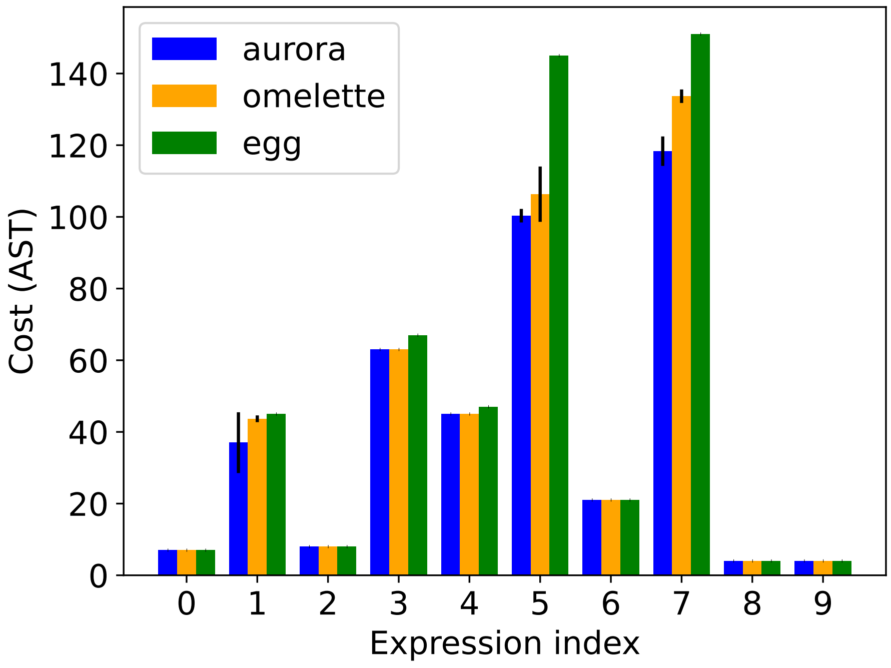


Figure 4.13: RL performance in the mathematical simplification domain (500 nodes upper bound; 3 seeds). The cost is the size of the Abstract Syntax Tree. Lower cost is better.

Figure 4.13 provides evidence that both methods can surpass the performance of egg in the mathematical simplification domain. Our approach outperforms Omelette in this compiler domain, similar to how it surpasses it in the field of databases, with Aurora exceeding its performance in expressions '1', '5', and '7', respectively. This shows that the quality of Aurora spans across multiple compiler languages and proves *Hypothesis 4*.

Chapter 5

Summary and Conclusions

In this research project, the various challenges underpinning the design of an equality saturation pipeline for query rewrite have been addressed. We have demonstrated *how* and *when* equality saturation can improve upon mainstream database optimisers. To this end, we have shown that equality saturation cannot efficiently achieve saturation even on queries of moderate complexity due to high planning latency and the potential for non-termination.

To address these challenges, we have introduced Aurora, a query optimiser that guides the application of rewrite rules within the equality saturation framework. By directing our attention to learning methods, we have devised an expressive reinforcement learning pipeline for solving the e-graph construction combinatorial problem, which can also be applied to other languages than SQL.

By experimentally evaluating the proposed methodology against Omelette, a neighbouring RL architecture for e-graph construction, and egg, the current state-of-the-art for equality saturation, we have demonstrated that Aurora outperforms them in terms of expressiveness, especially in real-world domains like query optimisations. Concretely, we have verified the following hypotheses:

1. *Aurora can rewrite complex database queries to significantly reduce the end-to-end latency*
2. *Aurora is generalisable across different database management systems*
3. *Aurora is able to find lower cost solutions than the leading methods for equality saturation such as Omelette and egg*
4. *Aurora provides unique advantages over Omelette in terms of convergence.*

Summary. We bring two major contributions with our research: 1) the proposed methodology is orthogonal to the existing query rewrite literature in the DB community, and in light of the experimental results, we deem it as a promising research avenue; and 2) we improve upon the current solvers for equality saturation in a complex real-world domain, with RL techniques generalisable across programming languages.

5.1 Future Work

1. **Generalisation** While we have shown the RL behind Aurora is promising in complex language domains, challenges remain. Adjacent to Omelette, the greatest open problem is generalisation. While the proposed methodology can generalise across unseen query rewrite problems in theory, expecting an agent to solve NP-hard problems across sparse problem instances (such as the space of possible SQL e-graphs) is not realistic. A promising extension to the framework is the adoption of population-based Reinforcement Learning (e.g. Poppy [17]), which comes with theoretical guarantees for solving combinatorial optimisation problems more efficiently. Adjacent methods include beam-search [9] and graph-search methods [49].

2. **Learning Extraction** One of the core practical limitations of equality saturation for optimising online systems is the time overhead it incurs. In the database domain, we have shown that most of the planning latency is linked to extracting the optimal expression from the e-graph. Additionally, we have demonstrated how to formulate the SQL plan extraction model as an integer linear program (ILP). To this end, we remark that the extraction problem is also of combinatorial nature. However, different from the e-graph construction task, it has structure. On the basis of the deluge of work associated with combining constraint-based problems with RL [3, 67, 7], we deem this is a promising future research opportunity. The aim is to learn the extraction routine rather than to solve new problem instances from scratch, in order to mitigate the time overhead associated with the procedure.

Bibliography

[1] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

[2] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pages 30–37. Elsevier, 1995.

[3] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In *International conference on machine learning*, pages 344–353. PMLR, 2018.

[4] 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 *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230, 2018.

[5] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

[6] Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-yan Liu, and Liwei Wang. Graphnorm: A principled approach to accelerating graph neural network training. In *International Conference on Machine Learning*, pages 1204–1215. PMLR, 2021.

[7] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3677–3687, 2021.

[8] Guillaume MJ B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*, pages 60–71. Springer, 2008.

[9] F Della Croce, Marco Ghirardi, and Roberto Tadei. Recovering beam search: Enhancing the beam search approach for combinatorial optimization problems. *Journal of Heuristics*, 10:89–104, 2004.

[10] Beatrice Finance and Georges Gardarin. A rule-based query rewriter in an extensible dbms. In *Proceedings. Seventh International Conference on Data Engineering*, pages 248–249. IEEE Computer Society, 1991.

[11] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[12] Morten Frydenberg. The chain graph markov property. *Scandinavian Journal of Statistics*, pages 333–353, 1990.

[13] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[14] Goetz Graefe and David J DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 160–172, 1987.

[15] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*, pages 209–218. IEEE, 1993.

[16] Rick Greenwald, Robert Stackowiak, and Jonathan Stern. *Oracle essentials: Oracle database 12c.* " O'Reilly Media, Inc.", 2013.

[17] Nathan Grinsztajn, Daniel Furelos-Blanco, and Thomas D Barrett. Population-based reinforcement learning for combinatorial optimization. *arXiv preprint arXiv:2210.03475*, 2022.

[18] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[20] Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. *The ICLR Blog Track 2023*, 2022.

[21] Shengyi Huang, Rousslan Fernand JulienDossa Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and Joao GM Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *The Journal of Machine Learning Research*, 23(1):12585–12602, 2022.

[22] HamidReza Kadkhodaei and Fariborz Mahmoudi. A combination method for join ordering problem in relational databases using genetic algorithm and ant colony. In

2011 IEEE International Conference on Granular Computing, pages 312–317. IEEE, 2011.

[23] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[24] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM (JACM)*, 29(3):699–717, 1982.

[25] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *VLDB*, pages 96–107, 1994.

[26] Feifei Li. Cloud-native database systems at alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[27] Guy M Lohman. Grammar-like functional rules for representing query optimization alternatives. *ACM SIGMOD Record*, 17(3):18–27, 1988.

[28] Kevin Loney. *Oracle database 10g: the complete reference*. McGraw-Hill/Osborne London, 2004.

[29] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[30] 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.

[31] 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.

[32] Ryan Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *arXiv preprint arXiv:1902.00132*, 2019.

[33] Larry R Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 5:64–67, 2001.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[35] M Muralikrishna et al. Improved unnesting algorithms for join aggregate sql queries. In *VLDB*, volume 92, pages 91–102. Citeseer, 1992.

[36] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[37] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Term Rewriting and Applications: 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005. Proceedings 16*, pages 453–468. Springer, 2005.

[38] James R Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[39] Chengzhe Piao and Chi Harold Liu. Energy-efficient mobile crowdsensing by unmanned vehicles: A sequential deep reinforcement learning approach. *IEEE Internet of Things Journal*, 7(7):6312–6324, 2019.

[40] Hamid Pirahesh, Joseph M Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. *ACM Sigmod Record*, 21(2):39–48, 1992.

[41] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[42] Behandelt PostgreSQL. Postgresql. *Web resource: http://www. PostgreSQL. org/about*, 1996.

[43] postgresql.org. Relational algebraic equivalence transformation rules. `https://www.postgresql.org/message-id/attachment/32513/EquivalenceRules.pdf`, 2022. Accessed: 10 10, 2022.

[44] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[45] Risinglight. Risinglight. `https://github.com/risinglightdb/risinglight`, 2022. Accessed: 10 01, 2023.

[46] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[47] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[48] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 435–446, 1996.

[49] Dennis Shasha, Jason TL Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–52, 2002.

[50] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[51] Abraham Silberschatz, Henry F Korth, and Shashank Sudarshan. Database system concepts. 2011.

[52] Zak Singh. Deep reinforcement learning for equality saturation. `https://www.cl.cam.ac.uk/~ey204/pubs/MPHIL_P3/2022_Zak.pdf`, 2022. Accessed: 06 09, 2022.

[53] Zak Singh. Deep reinforcement learning for equality saturation. `https://github.com/ucamrl/eqs.git`, 2022. Accessed: 06 09, 2022.

[54] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6:191–208, 1997.

[55] Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560*, 2019.

[56] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[57] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[58] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[59] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

[60] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. Spores: sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951*, 2020.

[61] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with program synthesis. *arXiv preprint arXiv:2202.10390*, 2022.

[62] 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 *Proceedings of the 2022 International Conference on Management of Data*, pages 94–107, 2022.

[63] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[64] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[65] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, 2021.

[66] 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.

[67] Tianyu Zhang, Amin Banitalebi-Dehkordi, and Yong Zhang. Deep reinforcement learning for exact combinatorial optimization: Learning to branch. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 3105–3111. IEEE, 2022.

[68] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[69] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2020.

[70] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. A learned query rewrite system using monte carlo tree search. *Proceedings of the VLDB Endowment*, 15(1):46–58, 2021.

Appendix A

Experimental Settings

A.1 Hyperparameters

Hyperparameter	**Value**
Number of parallel environments	32
Learning Rate	2.5e-4
General Advantage Estimation λ	0.95
Discount Factor γ	0.99
Number of Minibatches	4
Surrogate Clipping Coefficient	0.2
Gradient Clipping (Max Norm)	0.5

Table A.1: A subset of the hyperparameters used for RL training

A.2 PostgreSQL Plans

```
# Alternative
('Seq Scan on orders  (cost=0.00..52345.00 rows=652128 width=4) (actual
   time=0.098..1155.178 rows=651493 loops=1)',)
("  Filter: ((o_totalprice > '40000'::numeric) AND (o_totalprice >
   '40000'::numeric) AND (o_orderstatus = 'F'::bpchar))",)
('  Rows Removed by Filter: 848507',)
('Planning Time: 0.188 ms',)
('Execution Time: 1930.978 ms',)

# Optimal
('Seq Scan on orders  (cost=0.00..48595.00 rows=652128 width=4) (actual
   time=0.073..1033.012 rows=651493 loops=1)',)
("  Filter: ((o_totalprice > '40000'::numeric) AND (o_orderstatus = 'F
   '::bpchar))",)
('  Rows Removed by Filter: 848507',)
('Planning Time: 0.103 ms',)
('Execution Time: 1732.170 ms',)
```