Maliva: Using Machine Learning to Rewrite Visualization Queries Under Time Constraints

Qiushi Bai, Sadeem Alsudais, Chen Li, Shuang Zhao
Department of Computer Science, UC Irvine, CA 92697, USA
{qbai1,salsudai}@uci.edu,{chenli,shz}@ics.uci.edu

ABSTRACT
We consider data-visualization systems where a middleware layer translates a frontend request to a SQL query to a backend database to compute visual results. We study the problem of answering a visualization request within a limited time constraint due to the responsiveness requirement. We explore the optimization options of rewriting an original query by adding hints and/or doing approximations so that the total time is within the time constraint. We develop a novel middleware solution called Maliva based on machine learning (ML) techniques. It applies the Markov Decision Process (MDP) model to decide how to rewrite queries and uses training instances to learn an agent to make a sequence of decisions judiciously for an online request. We give a full specification of the technique, including how to construct an MDP model, how to train an agent, and how to use approximating rewrite options. Our experiments on both real and synthetic datasets show that Maliva performs significantly better than a baseline solution that does not do any rewriting, in terms of both the probability of serving requests interactively and query execution time.

1 INTRODUCTION
As a powerful way for people to gain insights from data quickly and intuitively, visualization is becoming increasingly important in the Big Data era. A common architecture to support data visualization has three tiers: a backend database, a middleware layer, and a user-facing frontend. The middleware translates a visualization request to a query (typically in SQL) to the database, and sends the query answers to the frontend to display. This architecture is widely used due to its benefits of supporting in-situ analytics at the data source, and utilizing the database’s built-in capabilities of efficient storage, indexing, query processing, and optimization. Responsiveness is critical in these systems [5, 9, 28], and a request needs to be served within a time budget, e.g., 500ms. This requirement is especially challenging when the data volume is large, and the user request has ad-hoc conditions on attributes of various types.

In this paper, we study the problem of answering visualization requests with a predetermined time constraint. We focus on middleware-based solutions, with the advantage that they treat the backend database as a black box without changes, and can leverage the computing capabilities to do in-situ analytics. We consider both rewritings that return exact results and rewritings that return approximate results. As a motivating example, consider a system that visualizes social media tweets on the US map with a time constraint of 500ms. Its backend database has a tweets table with attributes Content, Location, and CreateAt. Its backend database has a tweets table with attributes Content, Location, and CreateAt.

Equivalent rewriting options. Suppose a user asks for a spatial heatmap of tweets containing the keyword covid on the Thanksgiving day of 2020 in a region. The middleware creates a SQL query shown in Figure 1(a), which takes 3.35 seconds to execute. For this query, the physical plan generated by the database uses the keyword to access the inverted index on the Content attribute to retrieve candidate records, then filters them using the two other conditions. If we rewrite the query into an equivalent new query by adding a hint (Figure 1(b)), the rewritten query takes only 0.3 seconds, as the hint helps the database generate a more efficient physical plan, which uses the temporal filtering condition to access the B+ Tree index on the CreateAt attribute.

Approximation rewriting options. Figure 2(a) shows another visualization request on a larger region, which takes at least 4.28s for the database to run, no matter what hints we add. In this case, we rewrite the query by using random sampling, resulting in an approximated query that takes only 0.45s to run (see Figure 2(b)).

Why the database fails? For the query in Figure 1(a), there are many reasons the database can fail to generate an efficient plan. One is the estimation error of the query cost due to an underestimation of the keyword covid’s selectivity. The cost-estimation problem in optimizers is notoriously hard [30]. For example, in our experiments (Section 7), out of the 602 visualization queries that had at least one physical plan that could finish within 500ms, PostgreSQL failed to choose an efficient plan for 269 queries due to its cost-estimation errors. Although there are many higher-accuracy estimators such as [16, 34, 36, 46, 56, 67, 68], their higher estimation cost prevents them from being adopted by a general-purpose database. For example, for OLTP queries that need to be finished within milliseconds, spending tens of milliseconds for the cost estimation is unacceptable. Our key observation is
that for visualization applications where requests come with a predetermined time constraint, the middleware can be more opportunistic than the traditional database. For instance, it can spend tens of milliseconds to estimate the costs of multiple rewritten queries, then choose an efficient one to answer the request within the given time constraint.

Challenges. We may enumerate all possible rewritten queries by applying different hints to a given original query. We then use one of the aforementioned query-time estimators (QTE for short) to estimate the execution time of these rewritten queries and choose the most efficient one. There are several challenges in using this approach in the context of interactive visualization. (C1) A main challenge is that the cost of estimating the execution time of a rewritten query can be significant given a tight time constraint. For example, in Bao [34], estimating the execution time of all rewritten queries for one original query can take up to 320ms in their experiments. (C2) Another challenge is the uncertainty caused by the estimation error of the QTE, and the fact that the backend database may or may not follow the provided hints to generate a physical plan. (C3) The third challenge is quality. For queries without equivalent rewritten queries that can meet the time constraint, approximate rewriting options need to be explored. It is critical to maximizing the quality of the result while ensuring the query time is within the time constraint.

We address these challenges by introducing a novel machine-learning-based technique called Maliva, which stands for “Machine Learning for Interactive Visualization.” The technique formulates the middleware task as a Markov Decision Process (MDP). For a given time budget, we train an MDP agent to balance the planning time and the execution time of the rewritten queries. By learning from previous experiences, the MDP agent judiciously explores different rewriting options, so that the total time (including planning and query execution) is within the time limit. (W e address challenge C1 in Section 4.) Applying reinforcement learning, which inherently handles uncertainty, Maliva solves the issues of inaccurate time estimation and database ignoring the query hints when rewriting a query. (We address challenge C2 in Section 5.) By considering visualization qualities of rewritten queries in the reward design of the MDP model, Maliva makes the best effort to maximize the result’s quality while ensuring the query time is within the time limit. (We address challenge C3 in Section 6.) Our experiments show that Maliva has a much higher chance (70x more than the original query) to generate an execution plan such that the total time is within a time limit. Interestingly, it can also reduce query execution time. Both improvements show the significant benefits of adding learning capabilities to the middleware to support responsive visualization.

The rest of the paper is organized as follows. After formulating the middleware query-generation problem in Section 2, we give an overview of Maliva in Section 3. We present the details of this MDP-based solution, including its states, actions, transitions, and rewards (Section 4). We present how Maliva trains an MDP agent offline and uses it to generate a rewritten query online (Section 5). We generalize the MDP-based solution to be quality-aware by considering approximation rewriting (Section 6). Lastly, we report the results of a thorough experimental evaluation of Maliva to show its performance and benefits (Section 7).

1.1 Related Work
Visualization is a broad topic studied in many communities, and here we focus on efficiency-related works. A survey [14] summarized studies on interactive data analytics and visualization, and there are several recent studies on this topic [21, 23, 25, 49].

Approximate Query Processing (AQP). There are many techniques for computing approximate answers to queries [6, 11, 15, 26, 40, 44, 45, 47, 51, 62, 72, 73]. These approaches focus on developing approximation solutions to compute high-quality visualization. Existing solutions can be adopted as approximation rules in Maliva, such as Sample-Seek [11], which generates error-bounded visualization results by running queries on a small sample table.

Datacube-based approaches. Related studies include [9, 10, 20, 22, 27, 29, 39, 65]. In these approaches, the predefined cube intervals cannot support visualization queries with arbitrary numerical range conditions. The proposed Maliva system efficiently computes results for visualization queries with arbitrary conjunctive selection conditions.

Progressive visualization. There are solutions to show visualization results progressively [6, 8, 9, 13, 19, 38]. For instance, DICE [9] uses random and stratified samples to present an approximate result and then incrementally updates the result. These progressive visualization systems can adopt the proposed Maliva middleware to further optimize the intermediate queries to increase their efficiency.

Prefetching-based approaches. Techniques including [1, 4, 53, 69] accelerate visualization queries by prefetching or caching their results. For example, ForeCache [4] divides visualizations into tiles and prefetches them based on predicted user behaviors. Maliva is orthogonal to these techniques, and it can be adopted by them to further optimize the database queries.

Visualization using big data systems. These techniques use Hadoop, Spark, and Hive to support visualizations [6, 7, 12, 58, 70]. For instance, HadoopViz [12] and GeoSparkViz [70] use Hadoop and Spark to generate high-resolution visualizations. Their focus is on offline construction, not on an interactive visualization for queries with ad-hoc conditions. The proposed Maliva middleware technique is complementary to these solutions.

ML for visualization. The survey in [63] summarized studies of applying ML techniques to different stages during the whole visualization pipeline. Examples are [31, 64] for data cleaning and preparation and [18, 32, 50] for visualization recommendation.
Our proposed system focuses on applying ML techniques to solve performance issues at the middleware. ML-based query optimization has recently been used in database optimizers [24, 34–36, 54, 61, 71], selectivity estimation [16, 46], and cost estimation [56].

**Comparison with Bao.** The recent Bao technique [34] uses hints to generate optimized queries by modeling the optimization as a multi-armed bandit problem. In the training phase, Bao applies Thompson sampling to minimize the training time and maximize the accuracy of its neural-network-based query time estimator (QTE). In the online query processing phase, Bao uses a brute-force strategy that enumerates all the candidate query-hint sets, estimates the query time of all rewritten queries, and chooses the fastest query-hint set as the output. Bao assumes that the cost of estimating query time is negligible. This assumption is not valid in Maliva as our problem is responsive visualization with a stringent time constraint (e.g., 500ms). Maliva focuses on the online planning problem when a brute-force strategy is not acceptable. We compared Maliva with Bao in the experiments and the results show Maliva outperformed Bao in various metrics.

## 2 PROBLEM FORMULATION

**Visualization architecture.** We consider a typical three-tier data-visualization system that consists of a backend database, a middleware layer, and a frontend. For each frontend visualization request, let $Q$ be the original SQL query for the request. Let $\tau$ be a time limit that quantifies the expected responsiveness of the system. Ideally, we want the total delay, from the time the user submits a request to the time the result is shown on the frontend, to be within $\tau$. The original query $Q$ may not meet the time-limit constraint when the backend database cannot generate a physical plan that is fast enough. To solve this problem, Maliva rewrites $Q$ with two kinds of options: query hints and approximation rules. By adding a query hint to $Q$, Maliva can help the backend database generate an efficient physical plan that computes the result within the time limit. For expensive queries where no physical plan can meet the time limit, Maliva can add an approximation rule to the original query such that the backend database computes an approximate result to trade the visualization quality for responsiveness. Note that the proposed approach also works in a more general setting of approximate query processing (AQP) where a time constraint is given.

**Query hints.** A query hint in a database is an addition to the SQL standard that instructs the database engine on how to execute the query. For example, a hint may tell the engine to use or not to use an index (even if the query optimizer would decide otherwise) [17]. A query hint does not change the semantic meaning of the query, i.e., the result computed by the database engine with the hint remains the same. Databases such as AsterixDB [2], MySQL [41], Oracle [43], PostgreSQL [48], and SQL Server [55] support a variety of query hints. For example, in Figure 3(b), Maliva adds two hints `+ Index-scan(t CreateAt)` and `Nest-Loop-Join(t u)` to the original query. They suggest the engine to use the index on the `CreateAt` attribute to scan the table $t$, and do a nest-join loop on tables $t$ and $u$.

**Approximation rules.** An approximation rule is a method to rewrite the original SQL query to compute an approximate result, and the new query takes less time. There are various approximation rules available in database systems, such as adding a `Limit` clause, applying a SQL-standard `TableSample` operator on a table, or substituting a table with a smaller table randomly sampled from the original table. For example, in Figure 3(b), Maliva rewrites the original query by substituting the table tweets with a sample table `tweetsSample20` with 20% randomly selected records.

Now we formally define rewriting options, rewritten queries, and the query-rewriting problem.

**Definition 2.1. (Rewriting Option).** Let $H$ be a set of query-hint sets and $A$ be a set of approximation-rule sets. A rewriting option (“RO” for short) is a tuple $(h, a)$, where $h \in H$ and $a \in A$. Note that both $h$ and $a$ can be the empty set $\emptyset$.

For instance, the rewriting option in Figure 3(b) is a tuple with a query-hint set of “use the index on `CreateAt` and do a nest-loop join on $t$ and $u” and an approximation-rule set of “substituting the table tweets with the sample table tweetsSample20”.

**Definition 2.2. (Rewritten Query).** Given an original SQL query $Q$ and a rewriting option $RO$, a rewritten query (“RQ” for short) is a new SQL query generated by applying $RO$ onto $Q$. If $RO = (\emptyset, \emptyset)$, then $RQ = Q$.

For example, Figure 3(b) is a rewritten query for the original query in Figure 3(a).

**Query-rewriting problem.** Given a visualization request’s original SQL query $Q$ and a time limit $\tau$, we want to generate a rewriting option, such that the total time of the corresponding rewritten $RQ$, including planning and query execution, is within $\tau$ and the quality of $RQ$’s result is maximized. To quantify the quality, we assume a given visualization quality function $F$. Let $r(Q)$ be the result of the original query $Q$ and $r(RQ)$ be the result of the rewritten query $RQ$. Then $F(r(Q), r(RQ))$ computes the quality of $r(RQ)$.

In Sections 3, 4, and 5, we study the case of using query hints only (i.e., without changing query results). In Section 6, we study the case where approximation rules are also used.

## 3 MALIVA: ML-BASED QUERY REWRITING

We now introduce the middleware technique called “Maliva” to solve the aforementioned query-rewriting problem. We first give an overview of the technique, then use an example to explain the details.

**Overview.** As illustrated in Figure 5, Maliva rewrites the original SQL query to answer a visualization request within a time budget. It considers a predefined set of rewriting options, which we denote as $\Phi = \{RO_1, \ldots\}$. For each $RO_i$, the rewritten query is denoted as $RQ_i$. The set of candidate rewritten queries is $\Phi = \{RQ_1, \ldots\}$.
Figure 4: The Query Rewriter acts like an agent who makes a sequence of decisions to generate a rewritten query (with a total time ≤ 500 ms). At time 0, the agent considers rewritten query RQ1 due to its low estimation cost (estimated 25 ms, the actual 30 ms is updated once RQ1 is explored). After estimating its execution time (1,300 ms), the agent knows that RQ1 is not viable since the total time is longer than 500 ms. The estimation of RQ1 affects the costs for estimating RQ2 and RQ7. The agent explores RQ3 and then RQ7. With the estimated execution time being 300 ms and the elapsed time being 150 ms, RQ7 is decided as a viable rewritten query because the total time (450 ms) is within 500 ms.

Figure 5: Overview of Maliva.

Maliva has a Query Rewriter that enumerates possible RQs and uses a Query Time Estimator (QTE) to estimate the execution time of each of them. The Query Rewriter uses the best effort to choose an RQ such that the total time, including the planning process and query execution, is within the time budget $t$. Such an RQ is called viable. The middleware then sends the rewritten query to the database. The Query Results Handler sends the retrieved result to the frontend to visualize.

Naively enumerating all available RQs in $\Phi$ is computationally prohibitive due to two reasons. First, the cost of Query Time Estimator to estimate the execution time of a rewritten query is not negligible. For instance, in some cases it could take up to 70 ms [56] or 300 ms [67]. Second, the number of RQs increases exponentially when the number of applicable indexing choices increases. For example, consider a selection query on a table with filtering conditions on m attributes, and the database has an index on each attribute. The number of query-hint sets in $H$ would be $2^m$, since the database can use any subset of the $m$ indexes to do filtering and then intersect the record lists to compute the final result. Therefore, the Query Rewriter needs to balance the exploration time for query estimation and the execution time of each chosen RQ to find a viable RQ.

An example. Maliva views query rewriting as a Markov decision process (MDP) [57] and adopts machine learning (ML) to solve this problem. We use the running example in Section 1 to illustrate how Maliva uses an MDP agent to make a sequence of decisions to find a viable RQ. For simplicity, we assume the rewrite-options (RO) set to involve query hints only. We will generalize the technique to consider approximation rules in Section 6. As shown in Figure 4, a query has three selection conditions on three attributes, and each of which has an index. Suppose in the set $H$ of query-hint sets, each attribute has a query hint of using or not using the index. Thus, we have $2^3 = 8$ query-hint sets to choose from. The agent makes a sequence of decisions to estimate the execution times of several rewritten queries and find a viable one $RQi$ (that uses the indexes on all three attributes). Next, we present the details of this MDP-based technique.

4 MDP MODEL FOR ADDING QUERY HINTS

In this section, we present the details of using an MDP model in Maliva to solve the query-rewriting problem, and discuss how to implement the Query Time Estimator (QTE).

4.1 MDP Model for Query Rewriting

MDP [57] is a formalization of sequential decision-making problems where an agent learns to achieve a goal from interaction with an environment. At each time step, the agent is in a state $s$, and chooses an action $a$ available in state $s$. The environment transits the agent to a new state $s'$, and gives the agent a corresponding reward $R(s,a)$. To train an MDP agent is to find a good policy $\pi_s$ such that if the agent follows the policy to choose an action for each state, it maximizes the total reward in the end.

We use the MDP model to solve the query-rewriting problem. For simplicity, we first focus on the case where rewriting options do not contain any approximation rules, which means no rewritten queries have quality loss. We will generalize the technique to consider approximation rules in Section 6. Without considering quality loss, the MDP agent learns to maximize the chance of finding a viable rewritten query for a given visualization request. The agent takes a sequence of actions, and each action chooses an RQ to explore. That is, it asks the query time estimator (QTE) to estimate the execution time of the RQ. The agent chooses an RQ based on the current state, and considers the future cost it needs to pay and the execution time of RQs already explored. The agent gets a bonus if it finds a viable RQ, or a penalty if it runs out of time. In the offline phase, by analyzing queries in the training workload, the agent learns to maximize the chance to receive a bonus. In the online processing phase, given a new query, the agent decides which RQ to explore in each step to
receive a bonus in the end. Now we present the details of how to use MDP to model the process of choosing RQs.

**States.** A state represents the past decisions, based on which the agent decides an RQ to consider next. Suppose we are given a predefined set of $n$ ROs, i.e., $\Omega = \{R_1, \ldots, R_n\}$. Correspondingly, we have $n$ candidate RQs, denoted as $\Phi = \{RQ_1, \ldots, RQ_n\}$. A state is a vector

$$s = (E, C_1, C_2, \ldots, C_n, T_1, T_2, \ldots, T_n),$$

which includes three pieces of information, as shown in Figure 6. (1) The elapsed time ($E$) captures how much time we have spent. (2) The estimation cost ($C_i$) for each possible rewritten query $RQ_i$ captures how much time is needed for the agent to estimate its running time. Each $C_i$ is initialized with a rough estimation collected offline and updated during the online planning phase. Note that the MDP state does not require the initial $C_i$ values to be accurate, and a rough estimation from history statistics suffices. The actual estimation costs will be collected while the MDP agent processes a query, as will be described soon in the definition of Transitions. (3) $T_i$ is the estimated time for each already explored $RQ_i$. Each $T_i$ is initialized with a 0 value until it is filled with an estimated execution time.

**Example.**

```
| Elapsed time | Estimation cost | Estimated time | State |
|--------------|-----------------|----------------|-------|
| $E$          | $C_1$           | $T_1$          | $s = (E, C_1, C_2, \ldots, C_n, T_1, T_2, \ldots, T_n)$ |
```

**Figure 6: An MDP state in Maliva.**

We assume for each rewritten query $RQ_i$, collecting the physical plan and its statistics (e.g., cardinality and cost estimations of each operator) is done by the QTE, and its time is captured by the MDP model’s estimation cost ($C_i$). We assume the rewritten queries’ physical plans and statistics are not available to the MDP model. Thus, the proposed MDP model is general, and can be applied to any query shape with any predefined query-hint set. A natural question is that without the statistics of the explored RQs stored in the state, how can the MDP model make a good decision on which RQ to choose next? Our answer is that the execution time of a rewritten query implicitly captures the statistics of the physical operators (e.g., the cost of doing an R-Tree index scan on the Location attribute). By keeping the estimated execution time of each explored RQ in its state, the MDP model can learn the correlations of the execution times between different rewritten queries and make good decisions.

**Actions.** An action, denoted as $a$, is to explore an RQ next. For each RQ, the agent asks the QTE to estimate its execution time. Meanwhile, the agent needs to pay cost as it takes time for the QTE to extract query features, possibly by collecting online statistics from the database, and running the estimation model to do the estimation. In the running example, at time 0, the agent decides to explore $RQ_1$. It asks the QTE to estimate $RQ_1$’s execution time.

**Transitions.** A transition function defines how the environment computes the next state, given the agent’s action in the current state. Let the RQ considered by action $a$ in state $s$ be $RQ_i$. We define the transition function $T$ as follows. First, the QTE estimates the time of $RQ_i$, and we add the estimated time $T_i$ to the state. Second, the estimation costs for other RQs could change. In the running example, to estimate $RQ_1$ that uses the R-Tree index on the Location attribute, we need to collect the spatial filtering condition’s selectivity on the Location attribute. To estimate $RQ_2$ that uses both the inverted index on the Content attribute and the R-Tree index on the Location attribute, we need to collect the selectivity values of the filtering conditions on both attributes. After the agent takes the $RQ_1$ action, we update the estimation cost of $RQ_1$ to be the actual time it costs and update the estimated estimation cost of $RQ_2$ by excluding the cost to collect the selectivity value of the spatial filtering condition. As shown in Figure 7, after estimating the time of $RQ_1$, we add the estimated time 1,300ms of $RQ_1$ to the state, update the estimation cost for $RQ_1$ from the estimated 25ms to the actual 30ms, and update the estimation cost for $RQ_2$ from the previous estimated 90ms to the new estimated 60ms. Lastly, the estimation takes time $C_i$, and we add it to the elapsed time so far to indicate how much time the agent has spent to explore different RQs. Note that the $C_i$ is the actual cost of estimating $RQ_i$, which could be different from $C_i$ because $C_i$ is an estimated cost for estimating $RQ_i$.

**Rewards.** A reward function defines the agent’s immediate gain when it takes a particular action $a$ in a given state $s$. In our setting, consider two cases to compute the reward function. (1) The first case is when the agent is at an intermediate state where it still has time for planning but has not yet found a viable rewritten query. In this case, the agent should not be awarded or punished since it has not made a decision yet. Thus the reward value is 0. (2) The second case is when the agent is at a termination state where it decides the rewritten query $RQ_i$ runs it against the database, and collects the execution time $\hat{T}$ of $RQ_i$. In this case, the agent should be awarded if the total time (including both the planning time and rewritten query execution time) is less than the time budget, or punished if the total time is more than the budget.

The agent decides a rewritten query by considering three situations. The first one is that the agent finds an RQ to be viable based on the estimation of the QTE before running out of time. For example, in Figure 4, after spending 150ms for planning, the agent decides $RQ_2$ as the chosen rewritten query, since the predicted total time of $RQ_2$ is 450ms, which is within the 500ms budget. The second situation is when the agent uses up the time budget and has to stop planning. The third situation is when the
agent has exhausted all candidate rewritten queries and has to
decide which RQ to choose. In the latter two situations, the agent
chooses the fastest RQ known so far as the final decision.

Formally, suppose the generated rewritten query by the agent
is 𝑅𝑄 and the actual running time of query 𝑅𝑄 is 𝑇. Then the
reward function 𝑅(𝑠, 𝑎) is defined as follows,

\[ 𝑅(𝑠, 𝑎) = \frac{(𝑇 − 𝑠.𝐸 − 𝑇)}{𝑇}, \]  
where 𝑠.𝐸 denotes the elapsed time so far in state 𝑠. If the total
time 𝑠.𝐸 + 𝑇 is less than the time budget 𝑇, which makes 𝑅(𝑠, 𝑎)
positive, then the agent receives a reward. The faster the rewritten
query is, the larger the reward will be. On the other hand, if the
total time exceeds the time budget, which makes 𝑅(𝑠, 𝑎) negative,
then the agent receives a penalty. The slower the rewritten query
is, the larger the penalty will be. Thus, guided by the reward
function, the MDP model will learn to find an efficient rewritten
query as soon as possible.

4.2 Query Time Estimator (QTE)
Take the sampling-based QTE described in [67] as an example. It
first builds an analytical cost model (e.g., linear regression model),
and uses it to estimate the execution time of a rewritten query
by collecting its statistics online. Specifically, it estimates the
selectivity values of the query conditions by running count(*)
queries on a small sample table, provides the values as input
features to the cost model, and uses the model’s prediction as the
query’s execution-time estimation. There are also other possible
solutions in the literature [36, 56, 68] that can be used by Maliva.
Note that QTEs are the focus of this paper, and Maliva leverages
a given QTE intelligently to balance the planning time and the
query execution time.

5 TRAINING AND USING THE MDP AGENT
In this section we discuss how to train the MDP agent offline
in Maliva on a workload of visualization requests and use it to
generate a viable rewritten query online.

5.1 Training the MDP Agent
Suppose we have a workload of queries 𝑊 = [𝑞1, 𝑞2, . . . , 𝑞𝑛].
Our goal is to find an optimal policy 𝜋∗ such that for any query
𝑞𝑖 ∈ 𝑊, the agent following policy 𝜋∗ maximizes the chance to
generate a viable rewritten query. We adopt the deep Q-learning
algorithm [37] for finding an optimal policy for the MDP agent.
Its basic idea is to use a neural network (called Q-network) to
represent a policy 𝜋. Given an input of a state vector, the q-
network outputs a Q-value [66] for each possible RQ in the state.
A higher q-value means that the rewritten query is more likely to
be viable given the current information. Its training process includes
two main steps. The first step is to generate a set of experiences
by exploring different planning sequences for queries in the
workload repeatedly. Each experience is a 4-tuple (𝑠, 𝑎, 𝑠′, 𝑅𝑄),
meaning that taking action 𝑎 from state 𝑠 results in state 𝑠′ with
reward 𝑅. The second is to replay those experiences to update
the q-network’s weights gradually such that the q-network can
approximate the q-values of the optimal policy for each state-
action pair.

To apply the q-learning algorithm to solve our problem, we
first design a q-network architecture to represent the target policy
𝜋∗, then introduce an algorithm for training the agent to max-
imize the chance to generate viable RQs for the given workload 𝑊.
Next we provide more details in both steps.

![Figure 8: Q-network in Maliva for query rewriting.](image)

**Q-network for the query-rewriting problem.** A q-network
approximates a policy 𝜋 using a neural network, and different
weights of the network reflect different policy instances. As
shown in Figure 8, the q-network adopts a simple layered ar-
chitecture that includes one input layer, two fully connected
linear hidden layers, and a linear output layer. The two hidden
layers have sizes similar to the input layer. The number of nodes
in the output layer is the number of possible actions in the model,
i.e., possible RQs. All the hidden layers adopt “ReLU” [52] as the
activation function. Given a state 𝑠, the input of the network is a
vector of 𝑠 = (𝐸, 𝐶1, 𝐶2, . . . , 𝑞𝑛, 𝑇1, 𝑇2, . . . , 𝑇𝑛). The output is the
q-value 𝐸(𝑠, 𝑎) for each action 𝑎 in state 𝑠, where action 𝑎 can
be any rewritten query 𝑅𝑄 in 𝑅𝑄Set = {𝑅𝑄1, 𝑅𝑄2, . . . , 𝑅𝑄𝑛}.

**Training an MDP agent for query rewriting.** Algorithm 1
details how we train the MDP agent. To apply deep q-learning,
we generate the replay memory 𝑀 of experiences. For a given
visualization query workload 𝑊 = [𝑞1, 𝑞2, . . . , 𝑞𝑛], we generate
a set of experiences. Each experience is a 4-tuple

(𝑠, 𝑎, 𝑠′, 𝑅𝑄),

where the agent in a state 𝑠 estimates the time of the hinted
query represented by an action 𝑎 and observes the next state 𝑠′
with a reward 𝑅. Note that different queries can have different
optimal policies. Our goal is to learn an optimal policy for the
whole workload. We let the agent explore all the queries in the
workload 𝑊 in multiple iterations until the policy converges or
the number of runs exceeds a maximum threshold. In each
iteration, we shuffle the order of queries to reduce the bias caused
by earlier queries on the exploration direction of later queries.
For each query 𝑘 in 𝑊, we let the agent complete a sequence of
decisions. At each step, it selects an RQ to estimate. It pays the
cost to estimate the rewritten query’s execution time, transits
to the next state, and receives an immediate reward. The agent
repeats the process until it reaches a termination state (line 9) in
one of the three cases. The first case is when the estimated time
𝑇(𝑎) of the rewritten query in action 𝑎 suggests it is potentially
viable, i.e., 𝑠.𝐸 + 𝑇(𝑎) ≤ 𝑇. The second case is when the agent
runs out of time, i.e., 𝑠.𝐸 ≥ 𝑇. The third case is when the agent
has exhausted all possible RQs, i.e., 𝑝 = 0.

When the agent decides which RQ to explore at each step
(lines 12 to 14), we adopt an ε-greedy strategy [37, 59] to balance
between the exploration of RQs with uncertain values and the ex-
ploration of RQs known with high values. With an ε probability,
the agent chooses a random RQ that it has not been considered
To this end, we have generated a new experience tuple \((s, a, s', r)\) to compute an immediate reward \(r\) and update the new state \(s'\). We then call the reward function \(\mathcal{T}\), which is defined as:

\[
L = (Q^\pi(s, a) - y)^2.
\]

We keep updating the policy \(\pi\) until it converges, i.e., the total accumulated reward of the training workload \(w\) does not improve much in new iterations (e.g., less than 1%).

### 5.2 Using MDP to Rewrite Queries Online

After we train an MDP agent, the query rewriter utilizes the agent to generate a rewritten query for a new visualization query \(q\) online. Algorithm 2 shows the pseudo-code. Starting from an initial state \(s\), we use the trained policy \(\pi\) to compute the q-values for all the RQs and select the one with the highest q-value as the action \(a\) (line 5). We then estimate the running time of query \(a\) and transit to state \(s'\) (line 6). We compute the immediate reward \(r'\) for estimating RQ in \(a\) (line 7). If the action \(a\) is a potentially-viable RQ (line 9), we output the query \(RQ_i\) in \(a\) as the generated rewritten query. Otherwise, we run out of time for the remaining RQs (line 11). Then we select the rewritten query \(RQ_i\) with the minimum execution time estimated so far and output it. If neither cases happen, we repeat the above process.

#### Algorithm 2: Generating an RQ online

**Input:**
- A new query \(q\)
- A trained policy \(\pi\)
- A transition function \(\mathcal{T}\)
- A reward function \(\mathcal{R}\)
- A time budget \(\tau\)

**Output:**
- An RQ \(RQ_i\)

1. State \(s \leftarrow (0, C_1, C_2, \ldots, C_n, 0, 0, \ldots, 0)\);
2. Remaining set \(\rho \leftarrow \text{query } q\text{'s all possible RQs}\) \(\{RQ_1, RQ_2, \ldots, RQ_n\}\);
3. Reward \(r \leftarrow 0\);
4. While True do
   1. Select a query with the highest q-value predicted by \(\pi\)
      \[
a \leftarrow \arg\max_{RQ_i} Q^\pi(s, RQ_i);
      \]
   2. Estimate query \(a\) and transit to state \(s'\)
      \[
s' \leftarrow \mathcal{T}(s, a);
      \]
   3. Compute the immediate reward
      \[
r' \leftarrow \mathcal{R}(s, a);
      \]
   4. Remove query \(a\) from the remaining set \(\rho\)
      \[
\rho \leftarrow \rho - \{a\};
      \]
   5. If \(s.E + T(a) \leq \tau\) then
      1. Return \(RQ_i\) represented by \(a\);
      2. If \(s.E \geq \tau\) or \(\rho = \emptyset\) then
         1. Return \(RQ_i\) with the minimum execution time estimated in \(s\);
      3. End

Before (line 12). Otherwise, it selects an RQ that has not been explored with the highest q-value based on the current policy weights (line 14). We start with a high probability \(e\) of exploration and gradually decrease it to favor exploitation with the training progress.

Once an RQ is decided by the agent as an action \(a\), we call the transition function \(\mathcal{T}\) (Section 4.1) to transit the agent to the next state \(s'\) (line 16). We estimate the query \(a\)'s running time and update the new state \(s'\) by adding the estimated time for \(a\), adding the cost to the elapsed time, and modifying the costs of affected RQs. We then call the reward function \(\mathcal{R}\) (Section 4.1) to compute an immediate reward \(r'\) for estimating the RQ in \(a\) (line 17). To this end, we have generated a new experience tuple \((s, a, s', r')\), and store it in the replay memory \(M\) (line 18). When \(M\) reaches its capacity \(C\), we replace existing experiences in a FIFO manner.

After processing a query, we update the policy \(\pi\) following the original deep q-learning algorithm [37] (line 21). We sample a random subset \(M'\) of experiences from \(M\). For each experience tuple \((s, a, s', r')\) in \(M'\), we first compute the target q-value \(y\) of the state-action pair \((s, a)\) using the Bellman equation [66]. We then update the weights in policy \(\pi\) by minimizing the loss value \(L\) between the target q-value \(y\) and the current q-value, where \(L\) is defined as:

\[
L = (Q^\pi(s, a) - y)^2.
\]
6 APPROXIMATION REWRITING OPTIONS

In this section, we generalize Maliva by considering rewriting options with approximation rules. Recall that using a query-hint set to rewrite an original query $Q$ into an $RQ$ can help the database generate an efficient physical plan that computes the actual result without any approximation. However, for expensive queries where no physical plan can meet the time constraint, by applying an approximation-rule set to $Q$, Maliva can generate an $RQ$ that efficiently computes an approximate result within the time budget. We first extend the MDP model in Section 4 to consider approximation rules. We then discuss two approaches to applying the MDP model to implement a quality-aware query rewriter. The quality-aware query rewriter makes the best effort to generate a viable rewritten query and maximize the result’s quality. In the end, we discuss the trade-offs between the two approaches.

6.1 Quality-Aware MDP Model

Consider the case where the rewriting options contain both query hints and approximation rules. A rewritten query can return an approximate result with quality loss. We need to let the MDP agent learn to maximize the chance to generate a viable rewritten query and maximize the quality of the query result simultaneously. To quantify the quality of a rewritten query, we assume a given visualization quality function $F$. Let $r(Q)$ be the result of the original query $Q$, and $r(RQ)$ be the result of the rewritten query $RQ$. Then $F(r(Q)), r(RQ))$ computes the quality of $r(RQ)$. For example, suppose we use the Jaccard similarity function to measure the quality of an approximate result. Figure 9 shows that the quality of the scatterplot visualization result of an approximate rewritten query $RQ$ compared to the original query $Q$ is 0.76. Note that Maliva does not have restrictions on quality functions, and many functions can be used, such as VAS in [44] for scatterplots and the function of distribution precision in [11] for pie charts.

\[
\begin{align*}
  F(r(Q)) &= \text{Jaccard similarity} \\
  r(RQ) &= \text{Quality of the result of } RQ
\end{align*}
\]

Figure 9: The quality of $RQ$ compared to $Q$ using a Jaccard-based quality function as an example.

Reward function for a quality-aware MDP model. To achieve the goal of guiding the MDP agent to learn to maximize the chance to generate a viable rewritten query and maximize the quality of the query result simultaneously, we extend the definition of the reward function in Section 4. Recall that the learning goal of an MDP agent is to maximize the accumulative reward. In Section 4, once the agent decides a rewritten query, it receives a reward that reflects the query performance in terms of the total running time. Guided by the reward, the agent learns to generate a viable rewritten query quickly. Similarly, the MDP agent can also learn to quickly generate a viable rewritten query with a high result quality if the final reward reflects both the decided rewritten query’s efficiency and quality. The main idea is to combine the efficiency defined in Section 4 and the quality. The new reward function is a weighted summation of both. Formally, suppose the generated rewritten query by the agent is $RQ$ and the actual running time of query $RQ$ is $T$. Then the new reward function $R(s, a)$ is defined as follows:

\[
R(s, a) = \beta \frac{r - s \cdot E - \hat{T}}{\tau} + (1 - \beta) F(r(Q), r(RQ)).
\]

The term $\frac{r - s \cdot E - \hat{T}}{\tau}$ represents the efficiency of the rewritten query in terms of running time compared to the time budget. The function $F(r(Q), r(RQ))$ represents the quality of the $RQ’s result.

6.2 Quality-Aware Query Rewriter

Now we discuss how to apply the extended MDP model to implement a quality-aware query rewriter. We present the technical details of two approaches and discuss their pros and cons. We will show the evaluation results in Section 7.

One-stage approach. A natural idea is to replace the MDP model in Section 4 with the quality-aware MDP model. We let the MDP agent simultaneously consider query hints and approximation rules as rewriting options. By applying the new reward function combining both the efficiency of the rewritten query and the result’s quality, the MDP agent learns to maximize the chance of generating viable rewritten queries and maximize the quality.

Two-stage approach. A drawback of the previous approach is that the agent might miss a non-approximate viable rewritten query. To solve this problem, we consider a two-stage approach, with a main idea to let the MDP agent exhaust all candidate query hints first and then explore those approximation rules. In the two-stage approach, Maliva first runs the original MDP model, excluding the approximation rules. If the agent finds a viable rewritten query, it outputs the $RQ$ as before. If the agent exhausts all candidate $RQ$s without finding a viable one, and the elapsed
time has not exceeded the time budget \( \tau \), then we run the new quality-aware MDP model that considers the approximation rules to find a viable RQ.

When the planning time for the original agent is longer than the time budget, the two-stage approach reduces to the case described in Section 4. In this case, the one-stage approach is preferred since it can increase the chance of generating a viable rewritten query considering approximation rules. When the planning time for the original agent is relatively small compared to the time budget, the two-stage approach has the advantage of not missing any non-approximate viable rewritten queries.

7 EXPERIMENTS

We conducted experiments to evaluate Maliva. In particular, we want to answer the following questions: (1) How well does it rewrite queries to support visualization requests? (2) How well does it generalize to different numbers of rewriting options? (3) How well does it perform for different types of queries (e.g., single-table selection queries and multiple-table joining queries)? (4) How well does it generalize to different time budgets, unseen queries and other databases? (5) How does it compare with related solutions? and (6) How much is its training overhead?

7.1 Setup

Datasets. We used two real datasets and a synthetic one as shown in Table 1. The Twitter dataset included 100 million geo-located tweets in the US from November 2015 to January 2017. We kept the timestamp, geo-coordinate, text message, and several user attributes for each tweet in a tweets table. For the experiment on join queries, we used the tweets table and a users table. The former had a foreign key of "user_id" referencing the "id" in the latter. We used the geo-coordinate attribute as the output for visualization (e.g., choropleth map, heatmap, or scatterplot). The NYC Taxi dataset [42] included taxi trip records within three years from 2010 to 2012. The third dataset was generated from the TPC-H benchmark [60]. We used the line-item table as the fact table. The attributes we used for query selection conditions are shown in Table 1.

Query workloads. We generated random queries on each dataset for training and evaluation. Take Twitter dataset as an example.

![Image of two-stage MDP approach](image)

**Figure 11:** Two-stage MDP approach. After running the original agent that considers the 8 query-hint sets defined in Figure 4 without approximation rules, we cannot find a viable RQ. We then run the new agent with the quality-aware MDP model that considers all 8 query-hint sets combined with 3 approximation-rule sets (e.g., substituting the tweets table with 20%, 40%, or 80% sample tables), resulting in 24 rewritten queries in total. After spending extra time exploring a few RQs, the quality-aware agent chooses \( R_{Q21} \) as the final decision.

| Dataset   | Record (millions) | Size  | Filtering Attributes          | Output Attributes          |
|-----------|-------------------|-------|-------------------------------|---------------------------|
| Twitter   | 100               | 57GB  | text, created_at,             | id, coordinates           |
|           |                   |       | coordinates,                  |                           |
|           |                   |       | users_statuses_count,         |                           |
|           |                   |       | users_followers_count         |                           |
|           |                   |       | pickup_latitude,              |                           |
|           |                   |       | trip_distance,                |                           |
|           |                   |       | pickup_coordinates            |                           |
| NYC Taxi  | 500               | 146GB | extended_price,               | id, pickup_coordinates    |
|           |                   |       | ship_date,                    |                           |
|           |                   |       | receipt_date                  |                           |
| TPC-H     | 300               | 65GB  |                              | quantity, discount        |
|           |                   |       |                              |                           |

We first randomly sampled a set of tweets from the base table. For each tweet, we generated a query as follows. We chose the text, created_at, and coordinates attributes for the selection conditions in the query. We generated three conditions based on the values in the sampled tweet. For text, we randomly selected a non-stop word in the original tweet’s text message as the keyword condition. For created_at, we generated a temporal range condition with the value in the original tweet as the left boundary. We divided the maximum range on the created_at attribute in the base table into multiple zoom levels, and randomly selected a level to generate the length of the range condition. Suppose the maximum range on created_at had \( L \) days. We computed the maximum zoom level on created_at as \( Z = \lceil \log_2(L) \rceil \). If we randomly chose a zoom level from range \([0, Z] \), we computed the length of the query condition range as \( l = \max(L/2^z, 1) \). Similarly, for the coordinates attribute, we used the exact coordinates in the sampled tweet as the center. We randomly chose a zoom level and generated a spatial bounding box as the spatial range condition for the query.

In the experiments, we evenly divided generated queries into a training workload and an evaluation workload. We further divided the training workload into two parts, one with two thirds of queries for training the agent and the other one with one third of queries for validation. For all experiments, we used a hold-out validation strategy, i.e., we used a workload to train multiple MDP agents, and used a validation workload to choose a best agent to be evaluated on the evaluation workload. We report the performance on the evaluation workload in the each section. When evaluating different approaches, the “difficulty” of the queries in the evaluation workload played an important role. For example, if none of the physical plans of a query were viable, then no approach can generate a viable plan without approximation. On the contrary, if a query had many (e.g., five) viable physical plans, it was relatively easy for a method to find one. We further divided the evaluation workload into subsets of queries based on their difficulty. To measure the difficulty of a query, we define the following metric. Given a time budget \( \tau \), a query’s number of viable plans is \( \sum_{i=1}^{n} T(P_i) \leq \tau \), where the set \( \{P_1, P_2, \ldots, P_n\} \) contains an original query Q’s all possible physical plans generated by applying different candidate query hints, and \( T(P_i) \) is the execution time of plan \( P_i \). We changed the number of viable plans and evaluated different approaches on the same query workload with the same number of viable plans.

QTE implementations. We implemented two QTEs to evaluate the Maliva’s performance. 1) **Accurate-QTE**. To isolate the effect of estimation errors on the Maliva’ performance, we used the actual execution time of the hinted queries as the estimation, and
set up a unit cost parameter to represent the time of collecting the selectivity value of one filtering condition in a given rewritten query. Unless otherwise stated, we used 40 ms as the unit cost of collecting one selectivity value for the Accurate-QTE. 2) We also implemented the ML-based approximate-QTE as presented in Section 4.2. We used a random sample table [67] to estimate the selectivity values of query conditions. The selectivity values were used by the approximate-QTE’s ML model to estimate the execution time of queries.

Performance metrics. We used two metrics to evaluate the performance of different approaches. Recall that a generated rewritten query is “viable” if its total response time (including both the planning time and the querying time) is within a given time budget. The “viable query percentage” (VQP) of a solution was the ratio of viable queries over all the queries in the workload. The other metric was called “Average Query Response Time” (AQRT), which was the average total response time of all the queries in the workload.

Query-rewriting Approaches. We compared the proposed MDP-based approaches with three related methods, i.e., baseline, naive, and Bao [33]. MDP-based approaches included an MDP agent using an approximate-QTE, i.e., MDP (Approximate-QTE), and an MDP agent using an accurate QTE, i.e., MDP (Accurate-QTE). In the baseline approach, the middleware relies on the database optimizer to generate a physical plan for the original query. In the naive approach, we used the same approximate QTE as the MDP-based approach, but enumerated all possible RQs in a brute-force way, then chose the best RQ as the output. The third approach was Bao [33]. We used its open-source release [3] as the server, which provided interfaces for training the model and using the model to choose the best plan for a given set of query plans. Its original client, which was a PostgreSQL plug-in, did not support query hints for using a specific index, which were required by our visualization queries. To solve this problem, we implemented a new client in Python to support such query hints while keeping their server implementation.

In the experiments, we ran both the database and the middleware on the same AWS t2.xlarge instance with four vCPUs, 16GB RAM, and a 500GB SSD drive. We implemented the middleware in Python 3.6 and the neural network using Pytorch 1.7. We evaluated Maliva on both PostgreSQL and a commercial database.

7.2 Performance on Using Query Hints

We evaluated the performance of Maliva for only considering query hints in rewriting options (i.e., no approximations). For each dataset, we generated queries with three filtering conditions and set up the rewrite-option set with 8 query-hint sets, i.e., using or not using the index on each attribute. We varied the evaluation workloads with different numbers of viable plans, and collected the VQP and AQRT metrics for each approach. Table 2 shows the number of queries in the evaluation workloads.

Table 2: Number of queries in evaluation workloads.

| # of viable plans | 0 | 1 | 2 | 3 | 4 | ≥ 5 |
|-------------------|---|---|---|---|---|-----|
| Twitter           | 518 | 97 | 234 | 118 | 153 | 69 |
| NYC Taxi          | 408 | 91 | 146 | 13 | 181 | 3 |
| TPC-H             | 381 | 107 | 310 | 66 | 47 | 0 |

Figure 12 shows viable-query percentages (VQP) on the three datasets. The MDP-based approaches and Bao outperformed the baseline approach significantly, with MDP (Accurate-QTE) as the best. For example, on the Twitter dataset, for the queries with a single viable plan, both MDP-based approaches increased the VQP from the baseline’s 1% and Bao’s 20% to more than 70%. In most cases, MDP (Approximate-QTE) performed better than or comparable to Bao. In one case of the TPC-H dataset, Bao
performed better than MDP (Approximate-QTE) mainly because Bao’s QTE had a much higher accuracy than the approximate QTE for TPC-H. When the number of viable plans increased from 1 to 4, the VQP of all approaches increased because the more viable plans existed for a query, the easier it was for each approach to find a viable plan in a short amount of time.

Figure 13 shows the results of the average query-response time (AQRT) of different approaches. On the Twitter dataset, Bao had a comparable AQRT to the baseline, while MDP (Approximate-QTE) had much lower time than the baseline and Bao for queries with one or two viable plans. For example, MDP (Approximate-QTE) reduced the average response time from the baseline’s 1.11 seconds and Bao’s 1.01 seconds to 0.4 seconds. On the NYC Taxi dataset, Bao and MDP-based approaches had comparable performance and were slightly better than the baseline. On the TPC-H dataset, Bao was better than or comparable to the baseline. In two cases, Bao performed better than MDP (Approximate-QTE) because Bao’s QTE had a much higher accuracy than the approximate QTE on TPC-H. However, in all cases, MDP (Accurate-QTE) always had a lower query time than Bao and the baseline, which means it generated a more efficient plan. In cases where MDP (Accurate-QTE) had a longer response time, the extra planning time was the main reason. At the same time, the high VQP of MDP (Accurate-QTE) proved the ability of the MDP model balancing the planning time and the query-execution time to maximize the chance of generating a viable rewritten query.

7.3 Effect of Rewrite-Option Number

The number of rewrite options affects both the state and action space of the MDP model, which will further affect the performance of the MDP agent. We evaluated the effect of the number of rewrite options on the Twitter dataset. We set up two workloads of queries with filtering conditions on 4 attributes and 5 attributes of the tweets table, resulting in a rewrite-option set with 16 query-hint sets as rewrite options and the other rewrite-option set with 32 rewrite options. We compared the MDP-based approaches with Bao and the baseline approach. We varied the evaluation workloads with different groups of viable-plan numbers and collected the VQP and AQRT metrics of each approach. Table 3 shows the number of queries for the workloads.

| Workloads with 16 rewrite options. |               |               |               |               |               |
|-----------------------------------|---------------|---------------|---------------|---------------|---------------|
| # of viable plans                 | 0             | 1-2           | 3-4           | 5-6           | 7-8           | ≥ 9           |
| # of queries                      | 485           | 150           | 241           | 90            | 132           | 93            |

| Workloads with 32 rewrite options. |            |            |            |            |            |
|-----------------------------------|------------|------------|------------|------------|------------|
| # of viable plans                 | 0          | 1-4        | 5-8        | 9-12       | 13-16      | ≥ 17         |
| # of queries                      | 412        | 141        | 197        | 159        | 151        | 145          |

Figure 14 shows the VQP results of different approaches for workloads with 16 and 32 rewrite options. The MDP approaches (similar between using an Approximate-QTE and using an Accurate-QTE) performed the best in both cases, generating up to 40x more viable queries than both Bao and the baseline approach for the workload with 16 rewrite options on queries with one or two viable plans. As expected, the advantages of MDP-based approaches over the baseline approach became smaller when the number of rewrite options increased to 32. Estimating the query time became more expensive for queries with more rewrite options, and it increased the planning time of MDP-based approaches significantly compared to queries with fewer rewrite options.

Figure 15 shows the results of average query-response time. Consistent with the VQP results, all MDP-based approaches outperformed both Bao and the baseline approach for hard queries with fewer viable plans. For example, the MDP-based approach using an Approximate-QTE reduced the average query-response time from the baseline’s 1.13 seconds and Bao’s 1.05 seconds to 0.66 seconds for workloads with 16 rewrite options and one or
two viable plans. On the other hand, for queries with more viable plans, the MDP approach using an approximate-QTE had a comparable query-response time to Bao and the baseline approach. However, the MDP approach using an accurate-QTE always had a lower average query time than all other approaches. That is, the former generated a more efficient plan for most queries than the latter, but the total response time might be longer due to the extra planning time.

### 7.4 Effect of Time Budget

We evaluated the effect of time budget on the performance of different approaches. We varied the time budget to 0.25, 0.75, and 1.0 seconds on the Twitter dataset. For each time budget, we used different query workloads with different numbers of viable plans, and collected both the VQP and AQRT metrics for each approach.

Figure 16 shows the results of VQP for different time budgets. The MDP-based approaches outperformed both Bao and the baseline approach significantly for all the budgets. When the time budget was 0.25 second, MDP (Approximate-QTE) outperformed MDP (Accurate-QTE) significantly because the latter was too expensive for planning. On the other hand, when the time budget was 1.0 second, MDP (Accurate-QTE) outperformed MDP (Approximate-QTE) since the agent could afford the expensive estimation cost for more accurate estimations to find better-rewritten queries. These results show that the MDP model is adaptive to QTE’s with different costs and accuracies for different time budgets.

Figure 17 shows the results of the average response time for different time budgets. For queries with one or two viable plans, all MDP-based approaches outperformed both Bao and the baseline approach. For queries with three or four viable plans, MDP (Approximate-QTE) had a performance comparable to Bao and the baseline approach.

### 7.5 Performance on Join Queries

To evaluate the performance of Maliva on queries with joins, we set up a workload of queries joining the tweets and users tables with filtering conditions on three attributes. We compared the performance of the MDP-based approaches with Bao and the baseline approach. For the MDP-based approaches and Bao, we considered 7 different ways of using or not using indexes on the three attributes and 3 different join methods (i.e., nest-loop-join, hash-join, and merge-join) between the two tables. Thus we had 21 query-hint sets in total as the rewrite options. We varied the evaluation workloads with different groups of viable-plan numbers and collected each approach’s VQP and AQRT metrics.

Figure 18(a) shows that for all workloads of different viable-plan numbers, the MDP-based approaches outperformed Bao. For the queries with only one or two viable plans, MDP (Approximate-QTE) generated more than twice as many viable plans than Bao. Figure 18(b) shows that MDP (Approximate-QTE) outperformed Bao in all cases. For queries with one or two viable plans, the MDP-based approach reduced the average query response time from Bao’s 0.87 second to 0.34 second. Bao uses the plan tree and operators’ cost estimations from the physical plan generated by the underlying PostgreSQL database. Bao’s QTE also suffered from estimation errors where the database had wrong cardinality estimations. Maliva mitigated this problem by applying a more expensive sampling-based approximate-QTE. By judiciously choosing rewritten queries to run the expensive QTE, Maliva generated more efficient queries than Bao.

**Maliva vs Bao:** In general, the MDP-based approaches performed much better than Bao for workloads of queries with textual and spatial filtering conditions (e.g., Twitter and NYC). The reason was that Bao’s QTE relied on the plan tree and operators’ cost estimations from the physical plan generated by PostgreSQL. It also suffered from the large estimation errors by PostgreSQL for textual and spatial filtering conditions. Maliva mitigated this
problem by applying a sampling-based approximate-QTE that had better estimations for textual and spatial conditions. By judiciously choosing rewritten queries to run the expensive QTE, Maliva generated more efficient queries than Bao. We also evaluated the approach that used Bao with the expensive approximate QTE (labeled "Naive (Approximate-QTE)" in Figure 14(a)), and Maliva still outperformed Bao because of Bao’s long time of enumerating all possible rewritten queries.

Figure 19: Generalization to (a) handle unseen queries and (b) use a commercial database.

7.6 Unseen Queries and Other Databases

To evaluate how well Maliva can be generalized to handle unseen queries, we did experiments on the Twitter dataset to train and test the MDP model using two workloads with different query shapes. The training queries were on a single tweets table with three filtering conditions. In comparison, the testing queries joined the tweets table and the users table on user_id with three filtering conditions on the former table. As shown in Figure 19(a), the MDP-based approaches outperformed the baseline significantly on the workload with unseen queries. For example, for queries with a single viable plan, the MDP (Approximate-QTE) approach increased the VQP from the baseline’s 2% to 55%, and the MDP (Accurate-QTE) approach further increased it to 74%.

We also did experiments on the Twitter dataset using a commercial database. We used a smaller table with 10 million records and thus a smaller time budget (250ms). The result is shown in Figure 19(b). Due to the commercial database’s complex behaviors, the approximate QTE had a much lower accuracy (two orders of magnitude) than it had on PostgreSQL. The reason was the approximate QTE only considered predicates’ selectivities for estimation, but more factors in the commercial database affected the query time, such as buffering and dynamic execution plan change. However, MDP (Approximate-QTE) still had comparable performance (VQP) to the baseline. With a more accurate yet more expensive QTE, MDP (Accurate-QTE) outperformed the baseline for all the queries. For example, for queries with one or two viable plans, the baseline had a VQP of 23%, MDP (approximate-QTE) had a VQP of 36%, and MDP (Accurate-QTE) increased the VQP to 50%.

7.7 Performance of Quality-Aware Rewriting

We evaluated the performance of the two quality-aware query rewriting approaches (i.e., one-stage and two-stage) described in Section 6. We used them on the same Twitter dataset and workload as in Section 7.2. We compared them with the baseline approach and the MDP approach without considering approximation rules. For the quality-aware rewriting approaches, we considered five approximation rules (i.e., adding a LIMIT clause with 0.032%, 0.16%, 0.8%, 4%, and 20% of the estimated cardinality of the query) in addition to the eight query-hint sets (i.e., using or not using indexes on the three filtering attributes) considered in Section 7.2. All MDP-based approaches used an accurate-QTE. Besides the AQP and AQRT metrics, we collected a new metric called Jaccard-based Quality, which computed the Jaccard similarity between the visualization result of a rewritten query and that of the original query.

Figure 20(a) shows the VQP of these approaches. For the group of queries without any viable plan, the MDP-based approach without considering approximation rules and the baseline approach had a zero VQP. By generating approximate rewritten queries, the two-stage MDP-based approach increased the VQP to 24%, and the one-stage MDP approach further increased the VQP to 31%. There were 518 queries in the 0-viable-plan workload (Table 2), and the one-stage MDP approach generated more than 35 viable queries than the two-stage approach. As shown in Figure 20(b), the one-stage MDP approach also reduced the average query response time of the 0-viable-plan queries from the two-stage approach’s 1.21 seconds to 0.95 seconds. In terms of efficiency, the one-stage MDP approach outperformed the two-stage approach in all cases. Figure 20(c) shows the average Jaccard-based quality of the rewritten queries generated by different approaches. Both the baseline and the MDP-based approach without considering approximation rules had no quality loss. The two-stage MDP approach had a significant advantage.
over the one-stage approach in terms of quality. For example, The former increased the quality of the viable-plan queries from the one-stage approach’s 0.43 to 0.79.

7.8 Training Performance

We evaluated the training performance for workloads with eight, sixteen, and thirty-two rewrite options on the Twitter dataset. For each workload, we divided a set of about 1,400 queries into a training set and a validation set. Then we varied the number of training queries and randomly sampled those from the training set without replacement. We then used the sampled queries to train an MDP agent and tested its performance on both the training queries and the validation queries. We repeated the step ten times for each number of training queries and collected the mean and standard deviation of the VQPs. We conducted the experiments on the MDP-based approach using the Accurate-QTE. The unit cost for the three workloads was 100ms, 60ms, and 50ms, respectively, and the time budget was 0.5 seconds.

Figure 21(a) and Figure 21(b) show the trend when we varied the number of training queries. For the workload with eight rewrite options, the VQP on the validation set got close to the VQP on the training set for about 50 training queries, i.e., adding more training queries did not further improve the performance. For workloads with sixteen or thirty-two rewrite options, the validation VQP got close to the training VQP at about 80 and 150 training queries, respectively. For different sizes of the training set, we also collected the mean and standard deviation of the training time. Figure 21(c) shows the training time of different numbers of rewrite options on the training sizes. For the same number of training queries, more rewrite options resulted in a larger q-network, which took more time to update the weights. For the workload with thirty-two rewrite options, it took about 150 seconds to train an MDP agent on 150 training queries.

Remarks: The experiments show that Maliva outperformed the baseline approach in terms of both the number of viable queries and average query response time. Maliva generated up to 70x more viable queries than the baseline approach. The advantages of Maliva were shown in both the real datasets and synthetic dataset, for different numbers of rewriting options, different time budgets and different query workloads. Its offline training overhead was relatively small. By considering approximation rules considered, Maliva generated even more viable queries. The comparison with Bao shows the advantage of Maliva due to the fact these two techniques were designed with different settings and optimization goals.

8 CONCLUSIONS

In this paper we studied how to rewrite database queries to improve execution performance in middleware-based visualization systems. We explored two optimization options of adding hints and doing approximation. We developed a novel solution called Maliva, which adopts a Markov Decision Process (MDP) model to rewrite a visualization request under a tight time constraint. We gave a full specification of the solution, including how to construct an MDP model, how to train an agent, and how to use approximating rewriting options. Our experiments on both real and synthetic datasets showed that Maliva performed significantly better than the baseline without no-rewriting options in terms of both the probability of serving a visualization request within a time budget and query execution time.
