ReoptSMART: A Learning Query Plan Cache

Columbia University Computer Science Technical Report cucs-023-08

Julia Stoyanovich[a,*,1], Kenneth A. Ross[a,2], Jun Rao[b], Wei Fan[c], Volker Markl[b], Guy Lohman[b]

[a] *Columbia University, New York, NY 10027, USA*

[b] *IBM Almaden Research Center, San Jose, CA 95120, USA*

[c] *IBM Watson Research Center, Yorktown Heights, NY 10598, USA*

Abstract

The task of query optimization in modern relational database systems is important but can be computationally expensive. Parametric query optimization(PQO) has as its goal the prediction of optimal query execution plans based on historical results, without consulting the query optimizer. We develop machine learning techniques that can accurately model the output of a query optimizer. Our algorithms handle non-linear boundaries in plan space and achieve high prediction accuracy even when a limited amount of data is available for training. We use both predicted and actual query execution times for learning, and are the first to demonstrate a total net win of a PQO method over a state-of-the-art query optimizer for some workloads. ReoptSMART realizes savings not only in optimization time, but also in query execution time, for an over-all improvement by more than an order of magnitude in some cases.

Key words: Query optimization, machine learning.

[*] Corresponding author.

Email addresses: `jds1@cs.columbia.edu` (Julia Stoyanovich), `kar@cs.columbia.edu` (Kenneth A. Ross), `junrao@almaden.ibm.com` (Jun Rao), `weifan@us.ibm.com` (Wei Fan), `marklv@us.ibm.com` (Volker Markl), `lohman@almaden.ibm.com` (Guy Lohman).

[1] Supported in part by NSF grant IIS-0121239

[2] Supported by NSF grants IIS-0534389 and IIS-0121239

1 Introduction

Query optimization is central to the efficient operation of a modern relational database system. The query optimizer is typically invoked every time a new query enters the system. The optimizer identifies an efficient execution plan for the query, based on available database statistics and cost functions for the database operators. In commercial systems, great care has been taken to reduce the overhead of query optimization. However, the task of the optimizer is complex, and the join ordering problem alone has complexity that is exponential in the number of tables [18]. As a result, the cost of optimization may represent a significant fraction of the elapsed time between query submission and answer generation.

If identical queries are submitted, the database system can cache the optimizer's plan the first time, and avoid reoptimization for subsequent query invocations. The query processor merely has to check for syntactic identity of the query with the cached query. This idea can be generalized to queries with parameters. Constants in the query are replaced with "bind variables" to generate a query template, in which the bind variables are parameters. The query processor can then cache a plan for a query template rather than for a query. As a result, frequently-submitted queries that differ only in the constants can avoid the cost of query optimization. Oracle provides such a facility [1], as do DB2 [23] and Microsoft SQL Server [14].

There is a potential problem with this approach. A single plan is chosen for all instances of a query template. This plan, while optimal in a particular region of the parameter space, may be sub-optimal in another region. Savings achieved by not invoking the query optimizer may be nullified by the choice of a sub-optimal execution plan. In fact often the difference in cost between the optimizer's plan and the cached plan exceeds the optimization time.

Modern transaction processing systems are often required to handle thousands of transactions per second. Consider for example a web-based OLTP application, such as an on-line book store described by the TPC-W benchmark [3]. The system executes canned queries that share a small number of pre-defined templates, such as queries generated by the same HTML form, but differ in parameter values. An interactive system is expected to complete query processing and return results to the user in a short amount of time, often less than a second. A single user's queries may exhibit locality in the values of the submitted parameters, in which case a single query execution plan may be good enough. However, this locality is lost when many users interact with the system at any given time. Therefore, to ensure that an optimal plan is chosen for every query invocation, every instance of the query must be optimized anew. Many of these queries involve joins of several database tables and are thus non-trivial to optimize. In this setting, query optimization may add significant overhead to the overall execution time.

Parametric query optimization (PQO) models the distribution of plans chosen in different regions of the parameter space of a query template [16], or of a set of templates [13]. A PQO system is *trained* off-line using a number of invocations of the query optimizer on instances of the query template. The result of such training is a function that, given an instance of the query parameters, identifies a plan that is likely to be the optimizer's choice. To be useful, this function must execute quickly, much faster than the optimizer itself. The function must also have a compact representation, so that a collection of such functions can be managed in memory by the database system.

Hulgeri and Sudarshan [15,16] construct a geometric subdivision of the parameter space. During training, they explicitly construct a parameter space decomposition into convex regions corresponding to individual optimal plans. At run-time, when query parameters are known, an appropriate plan is chosen from the plan space. We extend this work by replacing the explicit geometric constructions of [15,16] with state-of-the-art machine learning techniques that analyze the training data and generate a set of classifiers that map parameter instances to plans.

Compared with earlier geometric approaches, machine learning techniques that we use in this work have the advantage of being effective with much less training data, and of automatically handling non-linear boundaries in plan space. Due to the compactness of the models, our classifiers have modest space requirements linear in the number of classes, and typically on the order of 10 KB per class. Our techniques apply for both qualified and categorical attributes of any datatype. We demonstrate experimentally that our methods accurately predict plans for uniform as well as for skewed data distributions. We demonstrate that the testing functions (i.e., identifying a plan given parameter values) can be performed in less than a millisecond per query. This is typically much cheaper than the cost of query optimization.

In our experimental evaluation we explicitly target queries for which ReoptSMART achieves a total net win over methods currently available in commercial database systems. During our evaluation we found that commercial optimizers choose an efficient query execution plan very quickly for most queries. We limit our attention to queries for which optimization constitutes a significant portion of the over-all time, and for which PQO can potentially make a difference. We quantify the performance of ReoptSMART in terms of both prediction accuracy and total optimization and execution time.

To our knowledge, we are the first to demonstrate a total net win of a PQO method compared to either choosing a single plan (without reoptimization) or reoptimizing each query instance. Additionally, we build in corrections to the optimizer by learning on observed query execution times, and improve performance by over an order of magnitude for some queries.

The rest of this paper is organized as follows. Section 2 provides the problem statement. We give a brief description of relevant Machine Learning concepts in Section 3 and describe how we apply these concepts to PQO in Section 4. Experimental evaluation is outlined in Section 5, followed by discussion of results in Section 6. We describe related work in Section 7 and conclude in Section 8.

2 Problem Formulation

We now formally define the problem to be addressed.

Definition 1 *A* bind variable *is a variable that can appear in a predicate within an SQL* `WHERE` *clause. A* query template *with d parameters is an SQL statement containing d bind variables, each occurring exactly once. Bind variables are ordered according to their occurrence, and named $b_1, \ldots, b_d$ respectively. We use $\vec{b}$ as shorthand for the d-dimensional vector $(b_1, \ldots, b_d)$.*

Definition 1 does not restrict the data type of parameters. They may be numeric variables, strings, or even user-defined types. These variables must appear in a `WHERE` clause; the `WHERE` clause may belong to the outer query block, or to a nested subquery.

Example 2.1 *The following query template has three parameters* `b1`*,* `b2`*, and* `b3`*.*

```
Select sum(price)
From Orders O1
Where O1.item = :b1 And O1.quantity < :b2 And
  Exists (Select *
          From Orders O2
          Where O2.item = :b3 And O2.date = O1.date)
```

Definition 2 *Let Q be a query template with d parameters and let $\vec{p}$ denote a d-dimensional vector of values (of the appropriate types) for those parameters. A* query $Q(\vec{p})$ *is the parameter-free SQL statement derived from Q by replacing each bind variable in $\vec{b}$ with the corresponding value from $\vec{p}$.*

Parametric query optimization involves finding good plans for many queries that are derived from the same query template. We abuse terminology slightly by referring to the regular optimizer's choice of plan as the *optimal* plan.

Definition 3 *One is given query template Q, and a set of historical queries $Q(\vec{p}_1)$, ..., $Q(\vec{p}_n)$ derived from Q according to some distribution of values $\vec{p}_i$. For each query $Q(\vec{p}_i)$, suppose that the optimal plan is P_i. The set of queries and their corresponding plans is called the* training set*, and n is the* size *of the training set.*

A parametric query optimizer *(PQO) has an* off-line *phase and an* on-line phase*. During the off-line phase, the PQO may read the training set and database statistics to generate some additional information I that is cached by the database. During the on-line phase, the PQO is given a previously unseen query derived from Q using the same parameter distribution that was used in the training set. The PQO is required to choose a valid plan for that query based on the current database statistics and I (but not the training set). The PQO is* correct *if the chosen plan is the optimal plan for the query.*

The PQO is permitted to return no plan, which means that it cannot identify the optimal plan with sufficient confidence.

An extended parametric query optimizer *(EPQO) operates on an extended training set that contains the set of optimal plans chosen by the optimizer for the queries in the training set, and, for each query, the actual execution time according to each optimal plan. The goal of the EPQO is to choose the plan with the smallest actual execution time.*

When the on-line phase returns a plan, the database system will typically execute that plan without explicitly calling the optimizer. When no plan is returned, the database system will either optimize the query using the regular query optimizer, or use some standard default plan. A parametric query optimizer can be measured according to several metrics.

- The time to generate I from the training set is called the *training time*. Since this is an off-line process, the training time does not have to be "interactive."
- The size of I represents the amount of space needed to be kept on-line by the database system to enable the PQO to optimize instances of a query template.
- The time taken during the on-line phase to identify a plan for a given vector of parameter values. Since the aim of PQO is to save the time taken by the query optimizer, this measure should be substantially faster than the regular optimizer itself.
- The on-line phase has three possible outcomes: correct plan, incorrect plan, and no plan. Note that the penalties for an incorrect plan and for no plan may be different. We may measure the extra time involved when a suboptimal plan is executed, and the extra time needed if the regular query optimizer is invoked.

These metrics may vary depending on the size of the database, the available statistics and access structures, the query template, the distribution of points in the parameter space, and the size of the training set.

While it is possible that there may be changes in database statistics between the off-line and on-line phases of PQO, we shall assume in this paper that the statistics remain valid between phases. If training is done sufficiently regularly, say each time the statistics themselves are recomputed, this is a fair assumption.

It is possible to perform parametric query optimization on the parameter values themselves. However, we choose instead to use the *selectivities* of the predicates involving those parameters as our inputs, for the following reasons:

(1) We know that the underlying optimizer bases its decisions on predicate selectivities, and not on any special property of values in the domain.
(2) We can leverage the existing optimizer to derive selectivity estimates, which are available for inspection after query optimization. Further, such statistics are available cheaply from the existing database statistics structures during the on-line phase.
(3) The approach is applicable to any data type for which the database system can estimate selectivities. [3]
(4) Different data values may map to the same selectivity measure. Using selectivity measures instead of actual values reduces the cardinality of the space and is the first step toward abstracting raw data into a model.

A well-known problem in query optimization is handling correlated columns. One solution is to use multidimensional statistics on combinations of columns for query optimization [19]. It is possible to do something similar for parametric query optimization by estimating a combined selectivity for groups of correlated predicates, and making this an additional input to the on-line and off-line phases. We leave handling correlated columns for future work.

3 Machine Learning Background

Machine Learning is the study of computer algorithms that improve automatically through experience [22]. Recent developments in this field have shown wide applicability of machine learning techniques [2,7,22].

A *classifier* is a computational procedure for deciding which among a number of classes an object belongs to, based on its properties. A binary classifier has

[3] User-defined data-types already need to provide selectivity estimation code if they want their types to be optimizable.

two classes: the positive examples and the negative examples. In a classification problem, objects are represented as labeled feature vectors $\{(\vec{x}_i, y_i)\}$ generated from a target true function $y = F(\vec{x})$, where $\vec{x}_i = (x_1, x_2, \ldots, x_k)$ is a list of features, and $y_i \epsilon Y$ is the class label. In our case the feature vector is a list of selectivity measures corresponding to the binding values for a query template, and labels are plans provided by the query optimizer. The task of *inductive learning* is to construct a model $y = f(\vec{x})$ to approximate the true function F.

In the interest of replacing the query optimizer function F with an inductive model f, we look for modeling techniques that are accurate in their prediction, and efficient in computation and in memory consumption during both model construction and query plan prediction. The error of a learning algorithm can be decomposed into bias, variance, and noise. We discuss each of these factors in turn.

There is no noise in our problem. We assume that the query optimizer function F is deterministic: given the same set of parameter selectivities for a particular query template, the optimizer will always return the same plan.

To achieve high accuracy, we need to choose an algorithm that can produce a function f that closely approximates the true function F. Such an algorithm is said to have low bias. Bias, or systematic error, quantifies the difference between the true decision boundary and the decision boundary constructed by the learning algorithm. For example, a stepwise single decision tree model will never be able to exactly represent a simple diagonal line. The lower the bias, the more closely the learned function approximates the true function. We excluded several machine learning techniques because of high bias. Regression techniques predict continuous values and are inappropriate for our domain in which there are clear discontinuities because the space of plan labels is discrete. The traditional single decision tree algorithm uses linear boundaries and is excluded from our study because the true decision boundary is non-linear in general. A wide variety of clustering algorithms are described in the machine learning literature. Clustering is typically used in unsupervised learning, where class labels are not available. However, PQO is a supervised learning problem – class labels are the query execution plans. We experimented with several other methods but found that they require extensive tuning and do not generalize across queries: Support Vector Machines (SVMs) were sensitive to the geometry of the plan space and required kernel selection; Naive Bayes was sensitive to the discretization (the way continuous features were mapped into buckets).

Variance quantifies the sensitivity of a learning method to the composition of the training set. An invariant algorithm is robust to variation in the training set, and to training set size, while a variant algorithm may construct functions

f that are quite different depending on the size and the composition of the training set. It has been shown that a reliable way to reduce variance is to construct multiple uncorrelated models and combine their predictions via some form of voting [28]. In this work we implement two algorithms, each using a different voting method, and compare their accuracy: AdaBoost [27] weighs the training data and constructs multiple classifiers from each sample, while Random Decision Trees [10] use a randomization approach. We compare bias and variance of the two algorithms in Section 5.6.

To summarize, we have considered a variety of machine learning methods, and excluded some of them because their modeling assumptions did not fit the problem at hand. We experimented with several other methods but found that they required extensive tuning and did not generalize across workloads. We focus our attention on AdaBoost and Random Decision Trees for this study. We are not aware of any other classification algorithms that are a good fit for this problem.

3.1 AdaBoost

Boosting is a general and provably effective method for improving the accuracy of any learning algorithm. AdaBoost [12] is a widely accepted boosting algorithm that can improve the accuracy of a collection of "weak" learners and produce an arbitrarily accurate "strong" learner. The weak learners are only required to be slightly better than random guessing, i.e., more than 50% accurate in the case of binary classification. AdaBoost was extended in [27] to handle *confidence-rated* predictions, where weak learners output both the predicted label and a confidence measure as their classification hypothesis.

AdaBoost calls each weak learner repeatedly in a series of rounds $t = 1, \ldots, T$. (There are various ways to choose T, and we'll describe one way in Section 4.1.) The main idea of the algorithm is to maintain a distribution of weights over the training set. Initially, the weights of all points are equal. On round t, each weak learner is measured by its *error* ϵ_t. The error is the sum of weights of (a) mis-classified points weighted by the confidence rating c of the prediction, and (b) correctly classified points weighted by $1 - c$. The weak learner with the lowest error is chosen; call it W_t. The weights of W_t's incorrectly classified examples are exponentially increased, and the weights of W_t's correctly classified examples are exponentially decreased. This way, the weak learners are forced to focus on the difficult examples in the training set. The process is repeated with the new weights. The final *strong hypothesis* H is the α_t-weighted majority vote of $W_1, \ldots, W_T$, where $\alpha_t = ln(\frac{1-\epsilon_t}{\epsilon_t})$. AdaBoost has provable bounds on generalization error, i.e., the error on unseen examples that come from the same distribution as the training examples [12].

Class	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	1	1	1	1	1	1
2	0	0	0	0	1	1	1
3	0	0	1	1	0	0	1
4	0	1	0	1	0	1	0

Fig. 1. ECOC for 4 classes.

AdaBoost is a binary classifier, while PQO is a multi-class problem. The basic AdaBoost algorithm has been extended to incorporate multi-class classification, and is known as AdaBoost.M2 [11]. We implemented this algorithm but failed to achieve fast convergence. We therefore looked for an alternative way to adapt AdaBoost to multi-class problems.

The simplest way to adapt a binary classifier to a multi-class problem is by using the "one-vs-all" approach, where a single classifier is built for every class. One-vs-all classification, while simple, is often unable to provide adequate prediction accuracy: there is no classification confidence measure, and if a point is classified positively by more than one classifier, no mechanism exists to break the tie.

The use of *error-correcting output codes (ECOC)* can improve prediction accuracy of binary one-vs-all classifiers on multi-class problems [8]. An ECOC is a matrix of binary values such as that in Figure 1. The length of the code is the number of columns in the matrix, and the number of rows corresponds to the number of classes in the learning problem. A single binary classifier, in our case AdaBoost, is trained for each column in the matrix, with points from classes that have a 1 in the corresponding entry serving as positive examples, and those from classes with a 0 entry – as negative examples. During testing, the incoming example is evaluated by every binary classifier, and a bit-string of classification outcomes is obtained. This string is then compared to every row in the matrix, and the Hamming distance (the number of bits that differ) is calculated. The point is assigned to the class closest in terms of Hamming distance. There is a trade-off between the improvement in prediction accuracy and training time: a greater Hamming distance can be achieved for longer codes, but more binary classifiers need to be trained.

3.2 Random Decision Trees

A *decision tree* is a classifier with a hierarchy of decisions made at each node of the tree. One traverses the tree from root to leaf, choosing the appropriate child based on the decision criterion coded into each node. For example, a node might have children for different ranges of the selectivity of the first predicate of a query template.

The Random Decision Tree (RDT) method constructs multiple decision trees "randomly." The construction selects a feature (in our case a predicate selectivity) at random from among those features not yet used in higher levels of the tree. A partitioning value for that feature is also selected at random from some distribution. Training data points from the node are then distributed to the node's children. Construction stops when the depth reaches a certain limit, when the number of data points in a node is sufficiently small, or when all points in a node have the same label (pure node). Unlike traditional single decision tree algorithms (e.g.,C4.5, ID3 [25]), RDT does not use gain functions to choose features and thresholds for tree nodes.

During the on-line phase, each tree is traversed using the actual query selectivities, to arrive at a leaf node L containing a number of plans. A posterior probability is calculated for each plan P. This probability is simply the proportion of the training points in L that are labeled with P. The posterior probabilities are averaged across all trees, and the plan with the highest average is output.

RDT has been shown to reliably estimate probabilities, closely approximate non-linear boundaries, and reduce variance when the number of training examples is small [10].

4 Our Approach

In this section we describe how we apply AdaBoost and Random Decision Trees, the two techniques that we refer to as *ReoptSMART*, to Parametric Query Optimization. ReoptSMART was implemented on top of an off-the-shelf commercial relational database, and requires no modifications to the optimizer or any other part of the database system.

4.1 Implementation of AdaBoost

We considered different ECOC lengths and observed that for c classes a length between $2 * c$ and $3 * c$ works well.

Choosing the weak learner appropriate for the domain was the main challenge of our AdaBoost implementation. Our choice of a weak learner was guided by the observation that the selectivity of a single parameter can be used to

```
Select i_title, a_lname, i_publisher
From Author, Item
Where a_id = i_a_id And i_stock < :b1 And i_page < :b2
```

Fig. 2. The Query Template TPCW-1

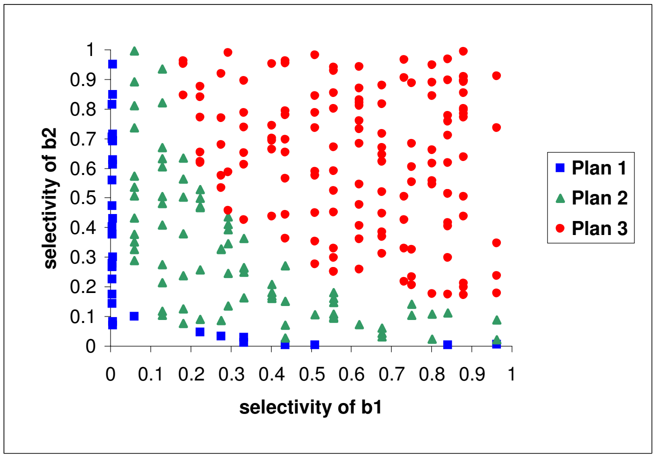

Fig. 3. Optimal plan space for query TPCW-1

discriminate between an optimal and a sub-optimal plan for a query in a particular selectivity region.

Consider the query in Figure 2 based on the TPC-W benchmark. The plan space for this query according to the DB2 Universal Database version 8.2 optimizer is represented in Figure 3. In this example the optimizer chooses the optimal plan based on the product of the selectivities of the two parameters. Plan 1 executes a nested loops join with the relation `Item` as the outer, and is chosen by the optimizer when the product of selectivities of $b1$ and $b2$ is very low. This happens when one or both of the selectivities are close to 0, and their product does not exceed 0.01. Plan 2 performs a hash join with `Item` as the build input. This plan is chosen for intermediate values of the two selectivities, with their product between 0.01 and 0.13. Plan 3 utilizes a hash join between the two relations with `Author` as the build input. This plan is optimal when both selectivities are higher than 0.2 and their product is above 0.13.

For queries with d parameters, the optimizer chooses a query execution plan based on individual selectivities and/or on products of any subset of the d selectivities. Products of selectivities naturally correspond to estimates of the relative size of intermediate or final results during plan execution. Explicit enumeration of all possible products (i.e. of all possible subsets of parameters) is exponential. We design our weak learners to avoid the exponential explosion and consider the selectivity of one parameter at a time. For Plan 1 we observe that the product of the selectivities is low if either one of the selectivities is less than 0.004, in which case the selectivity of the other parameter is immaterial, or if both $selectivity(b1)\epsilon[0, 0.06]$ and $selectivity(b2)\epsilon[0, 0.05]$.

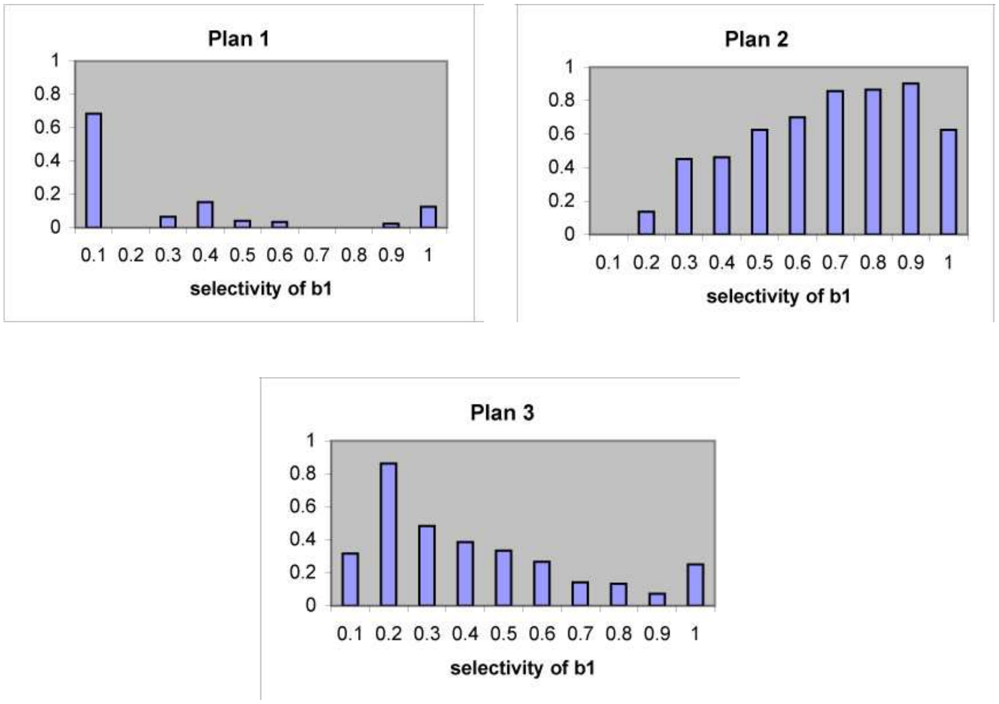

Fig. 4. Three weak learners for parameter $b1$.

The design of our weak learners is based on the above simple observation. Each weak learner is a discrete (bucketized) vector of weighted probabilities. The probabilities represent the likelihood that a particular plan is chosen by the optimizer when the selectivity falls within each bucket. The weights are adjusted over time by the AdaBoost meta-learner. A weak learner of this kind is defined for each parameter, and for each plan. Such weak learners are unary: they always claim that the point is a member of the class. They encode the strength of the claim in their confidence measure, which is proportional to the appropriate element of the weighted probability vector. The probability distribution is calculated using conditional probability:

$$Prob(plan|sel) = \frac{Prob(sel \wedge plan)}{Prob(sel)} \tag{1}$$

As is apparent from the above formula, we need only consider how many points that fall within the selectivity range of interest also map to the particular plan label. The initial distributions for Plans 1, 2 and 3 given selectivities of $b1$ are listed in Figure 4 for the data in Figure 3.

Our algorithm has two parameters that influence prediction accuracy: the number of training rounds T for each binary classifier and the number of buckets in the probability distributions B. We discuss each of them in turn.

AdaBoost adjusts the weights of correctly and incorrectly classified points

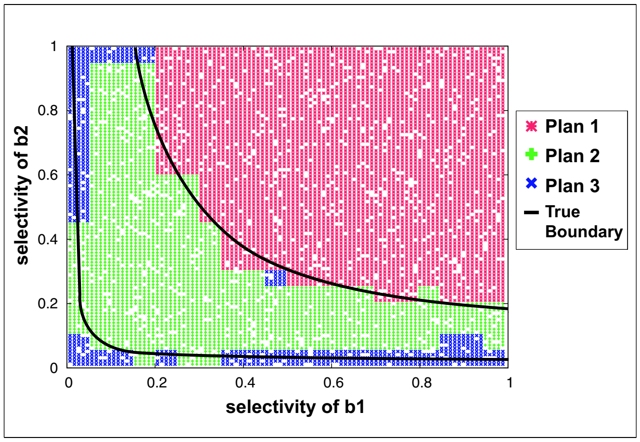

Fig. 5. AdaBoost decision boundary for TPCW-1

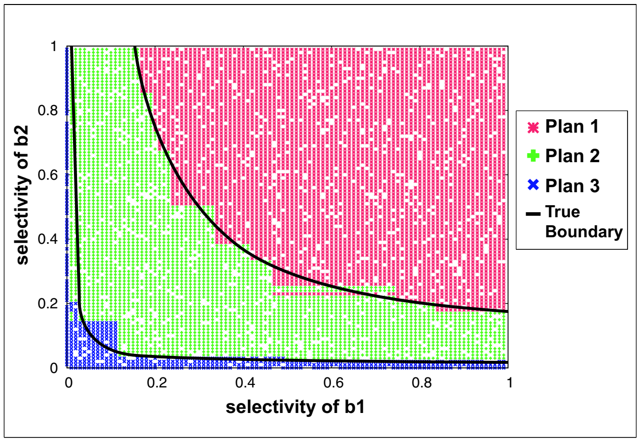

Fig. 6. RDT decision boundary for TPCW-1

exponentially, and provides for exponential convergence [12]. We train each binary classifier in increments of $T = 20$ rounds, and measure prediction accuracy with respect to the training set after each T rounds. We continue training until one of the following conditions is met: (a) $T = 100$ rounds or (b) accuracy on the training set reaches 95% or (c) accuracy on the training set did not improve compared to T rounds ago.

We use equi-width histograms with $B = 20$ buckets, each encompassing 5% of the selectivity range. This setting works well for all query templates in our experiments.

Figure 5 illustrates the decision boundary for query TPCW-1 learned by AdaBoost. The color of a region indicates the learned plan for that region.

4.2 Implementation of Random Decision Trees

To adapt RDT for query plan prediction, we make one important improvement based on our knowledge about the behavior of the optimizer. While predicates are still chosen at random, the decision threshold is no longer chosen at random. Instead, for a randomly chosen predicate, we compute a threshold with the highest information gain [22]. This way we are more likely to generate pure nodes, which leads to smaller trees. Our adaptation of RDT is similar to Breiman's Random Forest(RF) [6] but is more efficient. The RF algorithm uses computationally intensive bootstrap sampling (random sampling with replacement) from the training set, while RDT uses the original training set. Additionally, RF evaluates information gain for a set of features, while RDT considers a single feature at a time. Finally, RF uses voting to classify a point, and RDT uses averaged probabilities, which may benefit prediction accuracy, particularly for multi-class problems [10].

We choose the minimum number of examples per leaf node to be 2, which is the default in traditional decision tree methods. We construct 10 trees, due to a reported result that there is no significant improvement in accuracy when more than 10 trees are constructed [10]. We limit the depth of each tree to 5 times the number of features, which allows us to partition the range of selectivity measures into (up to) 6 ranges. Since the number of training examples is at most a few hundred, and no empty nodes are generated, each tree is expected to be reasonable in size.

Figure 6 presents the decision boundary learned by RDT for TPCW-1.

4.3 Improving Classification by not Making a Prediction

The accuracy of a classifier depends largely on the availability of sufficient training data. In many domains the number of training examples per class is highly non-uniform: some classes are represented by many more examples than others. During our experimental evaluation of AdaBoost we noticed that even on large training sets, the algorithm achieved higher prediction accuracy for the more common classes than for the less common. Not only were the points from the less common classes classified incorrectly, but a disproportionately large number of points was erroneously attributed to the less common classes.

It is often better to make no prediction for a test point than to classify that point incorrectly. For PQO, a misclassified point may incur a misprediction penalty that by far exceeds the optimization overhead. We looked for a reliable measure of prediction confidence that would direct the algorithm to give up on a point and generate an *uncertain* classification, i.e., “no plan.”

We first attempted to use the Hamming distance as a measure of prediction confidence. The algorithm was trained as before, but test points that fell outside the Hamming distance threshold of the closest class were classified as uncertain. We observed that previously correctly-classified points were now being classified as uncertain at approximately the same rate as the misclassified points, irrespective of the Hamming distance threshold. As a result, the overall prediction accuracy did not increase. We concluded that the mechanism for deciding the confidence of a prediction needed to be incorporated during the training phase, to ensure the proper generation of training classes.

During the training phase, all points that represent classes of size smaller than a threshold S are placed into a single *unreliable* class. We call S the *class size threshold*. The classifier is trained as before, but with fewer training classes.During the test phase, all points that map to the unreliable class are given the "no prediction" label. To determine a good class size threshold, we used cross-validation[4] on several datasets, and found that plans represented by fewer than 20 query instances were too small to classify reliably. We found that the class size threshold of 20 points worked well for all query templates and training set sizes, and we conclude that this parameter does not need to be learned with each new query template.

The training time of the algorithm is linear in the number of classes, and is reduced by grouping all uncommon plans together, as there are now fewer binary classifiers to train.

This technique reduces the misprediction rate by 5-10% for most queries, at the cost of some "no prediction" outcomes. For a mispredicted plan, the penalty depends on how suboptimal the chosen plan is. For a "no prediction" outcome, we expect to call the optimizer and then use an optimal plan. The penalty is therefore the cost of running the optimizer. The only way to compare a two-outcome classifier with a three-outcome classifier is to compute the total expected time with appropriate empirical measurements of the appropriate penalties. When we did so for AdaBoost, we found that the three-outcome classifier performed as well or better than the two-outcome classifier for almost all queries. We therefore only report experimental results for the three-outcome version of AdaBoost.

We did not implement a three-outcome version of RDTs primarily because the two-outcome version performed very well, as we shall see in later sections.

[4] The term *cross-validation* is used to describe the choice of some aspect of the machine learning model empirically, using available training data.

4.4 The Off-line and On-line Phases

Given a query template, and domains and distribution (such as uniform, Gaussian, or Zipf) of the query parameters, a training set of the appropriate size can automatically be generated for the query. For each point in the parameter space, the regular query optimizer is called and returns the query execution plan. (The query is not executed.) Alternatively, the real-time query workload may be recorded, along with the optimizer's predictions. This dataset is then used to train the classification algorithm. Training is computationally expensive and happens off-line. This is in line with how most query optimizers collect statistics.

For AdaBoost, during this phase each binary classifier executes between 20 and 100 training rounds T, and considers each available weak learner on every round. The number of binary classifiers is linear in the number of classes c, and the number of weak learners is exactly the number of query parameters d. Each weak learner computes probability distributions over the dataset, and is linear in the size of the dataset n. The time complexity of the training phase is therefore $O(T*d*c*n)$. The space complexity of the model is $O(T*B*c)$: each binary classifier returns T discrete probability distributions (each containing $B = 20$ entries), and T weights.

For Random Decision Trees, k random trees are independently constructed during the training phase. It has been reported in [10] that $k = 10$ returns satisfactory accuracy. The time complexity of training is $O(k * d * \log d * n * \log n)$. The output of the training phase has space complexity of $O(d*\log d*n)$.

When a new query matches an already-seen template for which the classifier has been trained, the instance will be classified on-line by the algorithm. If the classification outcome points to one of the learned classes, the plan for that class will be returned to the query execution module, and the query optimizer will not be invoked. If, however, an uncertain outcome is generated, the optimizer will be called upon to generate an execution plan. Optionally, query parameters and plan information can be stored for future re-training.

5 Experimental Evaluation

5.1 Experimental Setup

We evaluated the performance of our classifiers on the DB2 Universal Database version 8.2: a commercial RDBMS with a cost-based optimizer. All experiments were executed on a Pentium 4 3.0 GHz CPU, with 512MB of RAM, running the Linux operating system. The optimizer was executing at the highest optimization level. We chose to use level 9 to ensure that plans of highest possible quality are available for training. We decided that paying a higher optimization cost during the off-line phase was reasonable in order to realize full potential of ReoptSMART at run time.

AdaBoost and RDT were evaluated with respect to two database schemas. The first schema conforms to the TPC-W (web e-commerce) benchmark [3], with uniformly distributed data. We used two database sizes for our TPC-W experiments: 20MB and 100MB. Varying the size of the database resulted in a different number and types of optimal plans, and the shape of the plan space varied significantly. The second schema is the DMV database [21]: a synthetic database with skewed data distributions and correlations. We used a 60MB database for this schema. In our results, the *scale* of an experiment refers to the database size in megabytes.

The choice of relatively small databases is consistent with our target applications that demand very fast query response. When query execution time is small, optimization time becomes a significant component of the overall time.

The space overhead of ReoptSMART was no more than 30KB for all experiments.

5.2 Query Templates

We evaluate the performance of our algorithms with respect to 6 query templates. We choose templates for which optimization constitutes a significant portion of the overall time; these are the queries that can potentially benefit from Parametric Query Optimization. We considered many more templates, and the ones we choose highlight the cases where ReoptSMART makes the most difference. For many other queries, such as queries that are trivial to optimize or take a long time to execute, the opportunities for improvement were small. To show the applicability of ReoptSMART to a variety of query templates, we choose templates with a varying number and types of parameters, and with a varying number of joins.

The template TPCW-1 is given in Figure 2, the template TPCW-2 is given in Figure 7, and TPCW-3 is given in Figure 8. The template DMV-1 is shown in Figure 9. DMV-2 is shown in Figure 10 and contains an equality predicate on a categorical attribute `country`. Unlike numeric attributes, we do not expect nearby values of categorical attributes to have correlated behavior. Finally, DMV-3 is shown in Figure 11.

```
Select o_id
From Item, Order_Line, Orders
Where i_id = ol_i_id And ol_o_id = o_id
And i_page > :b1 And ol_qty > :b2 And i_subject = :b3
And o_status = :b4 And i_cost < :b5
```

Fig. 7. The Query Template TPCW-2

```
Select c_id, a_lname
From Customer, Orders, Order_Line, Item, Author
Where  i_a_id = a_id And  ol_i_id = i_id
And  ol_i_id = o_id And  o_c_id = c_id
And  i_subject = :b1 And  i_cost < :b2
And  c_birthdate < :b3 And  c_discount > :b4
Fetch First 10 Rows Only
```

Fig. 8. The Query Template TPCW-3

```
Select o.city, Count(*)
From Owner o, Demographics d
Where d.ownerid = o.id  And o.cars > :b1
And d.salary < :b2
Group by o.city
```

Fig. 9. The Query Template DMV-1

```
Select count(*)
From Owner o, Car c, Demographics c, Accidents a,
Location l, Time t
Where a.locationid = l.id And a.timeid = t.id
And a.carid = c.id And c.ownerid = o.id
And d.ownerid = o.id And t.year <= :b1
And d.salary = :b2 And o.country = :b3
```

Fig. 10. The Query Template DMV-2

For each of the above templates, we generated up to 1000 random parameter combinations. The distribution of each parameter matched the distribution of the column to which the parameter was compared. In some cases it was impossible to generate 1000 distinct parameter combinations because of the limited number of distinct values in the database. In such cases we generated the largest possible dataset. The training set of size 200 was chosen uniformly at random from the dataset; the remainder of the dataset was used for testing. Our measurements represent averages over 5 random splits of the dataset into training and testing sets.

A successful machine learning technique achieves high prediction accuracy with a limited number of training examples. We studied the effects of training set size on accuracy and observed only a 0-3% improvement in classification accuracy for RDTs when the size of the training set is increased beyond 200

```
Select count(*)
From Owner o, Car c, Accidents a, Location l, Time t
Where a.carid = c.id And a.locationid = l.id
And a.timeid = t.id And c.ownerid = o.id
And c.year = :b1 And t.year = :b2
```

Fig. 11. The Query Template DMV-3

Template	**Scale**	**Plans**	**Common Plans**	**Coverage**
TPCW-1	20	3	3	100%
TPCW-1	100	4	2	90%
TPCW-2	20	24	2	49%
TPCW-2	100	23	2	43%
TPCW-3	20	7	2	78%
TPCW-3	100	10	3	85%
DMV-1	60	9	3	87%

Fig. 12. Plan Space of Query Templates

points. For RDTs, all plans were considered during training and testing. Our current implementation of AdaBoost never chooses a plan if that plan is represented by fewer than 20 points in the training set. In our experience, increasing the size of the training set reduces the proportion of unclassified points, but does not significantly change the ratio of correct to incorrect classifications among the points for which classification was attempted. Increasing the size of the training set would enhance the performance of AdaBoost. However, we choose to present our results on just this limited training set because (a) it is less time-consuming to train on a small training set, and (b) for larger training sets, we observed only a marginal improvement in classification accuracy for RDTs. For a fair comparison we keep the training set size constant, and use 200 training points for all experiments.

The accuracy and overall performance improvement achieved by ReoptSMART for the first 4 query templates is presented in Sections 5.3 and 5.4. The final two templates, DMV-2 and DMV-3, will be discussed in Section 5.5.

5.3 Learning from Optimizer Predictions

In the first part of our experiments, we trained our classifiers on the plan space induced by the query optimizer, assuming that the query execution plan chosen by the optimizer is in fact the optimal plan for any given query

		AdaBoost			**RDT**	
Template	**Scale**	**Correct**	**Incorrect**	**No Plan**	**Correct**	**Incorrect**
TPCW-1	20	86%	14%	0%	92%	8%
TPCW-1	100	77%	9%	14%	93%	7%
TPCW-2	20	36%	10%	54%	73%	27%
TPCW-2	100	31%	6%	63%	76%	24%
TPCW-3	20	57%	19%	24%	91%	9%
TPCW-3	100	58%	23%	19%	89%	11%
DMV-1	60	76%	7%	17%	96%	4%

Fig. 13. Prediction Accuracy on Optimizer's Labels

		Total Time(sec)				**ADB % Improvement**		**RDT % Improvement**	
Template	**Scale**	**OPT**	**AVG**	**ADB**	**RDT**	**OPT**	**AVG**	**OPT**	**AVG**
TPCW-1	20	0.18	0.062	0.059	0.059	67	5	67	5
TPCW-1	100	3.9	1.90	2.1	1.8	46	-11	54	5
TPCW-2	20	11	14	11	9.7	0	21	12	31
TPCW-2	100	81	110	83	78	-2	25	4	29
TPCW-3	20	30	0.25	7.3	0.27	76	-2820	99	-8
TPCW-3	100	35	15	20	5.0	43	-33	86	67
DMV-1	60	190	200	180	180	5	10	5	10

Fig. 14. Performance Improvement on Optimizer's Labels

instance. Figure 12 summarizes some of the characteristics of the plan space for all query templates. The third column shows the number of distinct query plans that were generated using 200 training queries. The fourth column shows how many of those plans met the AdaBoost threshold that requires a plan to be chosen at least 20 times out of 200. The last column shows the fraction of the training points that employ plans that are above threshold. For example, for query DMV-1, 87% of the 200 training points optimized to one of the 3 plans that are above threshold.

Our accuracy results are summarized in Figure 13. This table indicates how often each of the classifiers was correct. For AdaBoost (to which we refer as ADB), we also include the frequency with which the algorithm made no prediction. Note that the "no prediction" column values are close to the proportion of queries *not covered* by plans that are above threshold. Our results indicate that Random Decision Trees achieve higher prediction accuracy on this domain compared to AdaBoost in our implementation. Both algorithms misclassify roughly the same portion of the dataset, but AdaBoost often makes no prediction, while RDT attempts to classify every point, so more points are classified correctly by RDT overall.

The classifiers choose a suboptimal plan some of the time. Nevertheless, it is possible that this plan has performance comparable to the optimal plan. As demonstrated in Figures 5 and 6, both AdaBoost and RDT closely approximate the true decision boundaries between classes, and any mispredictions will fall close to the boundary. In the remainder of this section we investigate how much worse a suboptimal plan can be, and to what extent the misprediction penalty can be offset by the reduction in optimization overhead. We compare the performance of our classifiers with two alternatives: (a) the cost of the plan generated by the optimizer plus the optimization overhead, for every query (OPT) and (b) total execution time of every query according to every plan in the optimal plan space, weighted by the frequency of each plan, plus the cost of a single query optimization(AVG). The second alternative represents the situation where a single plan is chosen for all instances of a query template. Typically the first plan that is encountered by the system is chosen, and the probability of any particular plan to be encountered first is approximately proportional to the frequency of that plan in our dataset.

We could not easily ask the commercial system we were using to estimate the cost of an arbitrary plan on different parameters. Thus, we measure the actual execution time of each query. We also discovered that, given a single query template, optimization time may vary depending on the plan that is ultimately chosen by the optimizer. This is because the choice of a plan affects how many plans are pruned at intermediate stages of plan generation. Therefore we do not assume that optimization time is constant for a given query template, and measure the optimization time per plan.

Figure 14 shows the improvement achieved by the machine learning methods compared to OPT and AVG. The improvement is calculated as $100\% * (T_c - T_m)/T_c$, where T_c is the time taken by one of the currently available methods, i.e., OPT and AVG, and T_m is the time taken by one of the machine learning methods. Positive values correspond to cases where machine learning methods outperform the currently available methods. All values in the tables are cumulative and represent totals over the entire test set.

Total times for both RDT and AdaBoost include the execution times of all points in the test set according to the chosen plan plus the overhead of the classification algorithms, which is on the order of 10 microseconds per point for both algorithms. AdaBoost also includes the optimization time for the points for which an uncertain prediction was generated, and the cost of the optimal

			Execution Time(s)		
Template	**Scale**	**Queries**	**HYBRID**	**OPT**	**Margin**
TPCW-1	20	50	0.059	0.059	0%
TPCW-1	100	800	1.8	1.8	0%
TPCW-2	20	300	9.4	9.5	1%
TPCW-2	100	400	25	76	67%
TPCW-3	20	800	0.19	0.27	30%
TPCW-3	100	800	0.45	4.8	91%
DMV-1	60	800	160	180	11%

Fig. 15. Predicted (OPT) vs. Observed (HYBRID) Optimal Plans

plan is used in this case.

As is apparent from Figure 14, AdaBoost outperforms OPT in most cases, while RDT outperforms OPT in all cases. Furthermore, the improvement achieved by RDT is always at least as high as that of AdaBoost, for two reasons: (1) RDT has a lower rate of mispredictions for most templates, and so suffers less of a misprediction penalty; (2) RDT never returns an uncertain prediction and suffers no optimization overhead. AdaBoost did not outperform the OPT method for TPCW-2 on the 100MB database because it did not attempt to classify enough of the space to offset the misprediction penalty by the reduction in optimization overhead.

It appears more difficult to outperform AVG for some queries. Despite high prediction accuracy on TPCW-3, even RDT performs 8% worse than AVG. This query has the highest optimization overhead: optimization takes a total of 29.5 sec, and execution – a total of 0.3 sec for 800 queries in our test set. Furthermore, there is no significant difference between the plans – executing all queries according to any single plan differs only marginally from optimal execution.

There is another important reason that limits the performance improvements realized by ReoptSMART. When training on the plan space induced by the optimizer, we trust that the plans chosen by the optimizer are indeed optimal. This, however, is not always the case. We explore a method for correcting the optimizer in the following section.

		AdaBoost			RDT	
Template	**Scale**	**Correct**	**Incorrect**	**No Plan**	**Correct**	**Incorrect**
TPCW-2	100	59%	12%	29%	69%	31%
TPCW-3	20	55%	17%	28%	64%	36%
TPCW-3	100	70%	16%	14%	75%	25%
DMV-1	60	66%	21%	13%	82%	18%

Fig. 16. Prediction Accuracy on Observed Labels (HYBRID)

5.4 Learning from Observation

The task of a relational optimizer is to choose a reasonably good plan among an exponential number of query execution plans. The optimizer does not explore the entire plan space, and brings no guarantee of global optimality. Additionally, the optimizer is guided by statistics that, even if accurate, represent only a summary of the data distributions and selectivities. For frequently executed short-running queries, it may be possible to build in "corrections" to the optimizer based on past execution time data. This is the Extended Parametric Query Optimization (EPQO) version of the problem as per Definition 3.

In the remainder of this section, we refer to query execution plans returned by the optimizer as *optimal* plans, and to plans that were observed to perform best as *best* plans.

To check how good a job the optimizer is doing, we look at actual query execution times for every query according to every plan that was optimal in some region of the selectivity space. We observe that this method introduces some noise: executing a particular query according to the same plan several times may yield slightly different execution times, especially if queries are fast-running. Additionally, for a particular query the best plan may perform only marginally better than the optimal plan. If we re-label all points with best labels, the resulting plan space is highly irregular and very difficult to learn. We use a simple method for reducing the noise: we only re-label a point with its best label if the best plan outperforms the optimal plan by more than 30%. We refer to the new labeling as *HYBRID*.

Having re-labeled the space, we compare the total execution times of all queries in the training set according to the optimal labeling with the execution times according to the hybrid labeling. These results are summarized in Figure 15. For TPCW-1 and for TPCW-2 on the 20MB database, the optimizer is making accurate predictions. However, the rest of the templates have some potential for improvement, and we focus on these templates in the remainder of this section.

		Total Time(sec)				**ADB % Improvement**		**RDT % Improvement**	
Template	**Scale**	**OPT**	**AVG**	**ADB**	**RDT**	**OPT**	**AVG**	**OPT**	**AVG**
TPCW-2	100	78	110	29	27	63	74	65	75
TPCW-3	20	30	0.25	5	0.25	83	-1900	99	0
TPCW-3	100	35	16	5	0.50	86	69	99	97
DMV-1	60	190	200	170	170	11	15	11	15

Fig. 17. Performance Improvement on Observed Labels

Prediction accuracy of AdaBoost and RDT on the test sets is summarized in Figure 16. Prediction accuracy on the observed labels (HYBRID) is lower than the accuracy on the labels generated by the optimizer (OPT). The reason for this is still the presence of noise, which makes the shape of the plan space more difficult to learn. In this noisy space, not making a prediction is starting to benefit AdaBoost.

Figure 17 summarizes the overall performance improvement of AdaBoost and RDT over OPT and AVG. We observe that the HYBRID re-labelling benefits all query templates. RDT now performs the same as AVG on TPCW-3 at scale 20, but outperforms both OPT and AVG in all other cases. AdaBoost still does significantly worse than AVG for TPCW-3 on the 20MB database, but outperforms OPT and AVG in all other cases. The performance improvement of both algorithms is more significant than when the classifiers were trained on the OPT space.

Based on our results, we conclude that machine learning techniques can benefit from learning on the HYBRID space. Generating HYBRID involves optimizing and executing each query in the training set according to every plan that is optimal in some region of the plan space. This is much more expensive than generating OPT, which does not involve any query executions, but is not prohibitively expensive for our target workloads of fast-running queries.

The HYBRID space can be used to improve the performance of ReoptSMART. It can also be used to determine whether to use ReoptSMART, or to simply optimize once and execute all queries according to a single plan, and perhaps suggest a good candidate plan. This will work best for queries for which there is no significant difference between individual plans, such as TPCW-3 on the 20MB database.

5.5 Queries with a Sparse Plan Space

We now turn our attention to query templates DMV-2 and DMV-3. For queries that utilize only, or mostly, equality predicates, many different parameter values will map to the same set of selectivities. In the extreme case, for a query that contains only equality predicates on uniformly distributed attributes, all points will map to a single set of selectivity measures, which in turn maps to a single optimal plan. We observed that 165 unique points were present in the selectivity space of DMV-2, mapping to 9 plans. For DMV-3, 24 unique points represented 6 distinct plans.

For such queries, a much simpler method to construct a query plan cache can be used. With a sufficiently large training set, not only all classes, but also all points, will be represented in the training set. A hash table that stores the mapping between the selectivities and the optimal plans will suffice; no learning is required, since the algorithm will never have to deal with any previously unseen selectivities.

However, it is difficult to tell if the training set is exhaustive by looking at that set alone. In this final set of experiments we show that ReoptSMART can be used for queries such as DMV-2 and DMV-3. In both cases, we train RDT and AdaBoost on all available points, and return the prediction accuracy on the training sets.

RDT achieves very high prediction accuracy on both sets: 163 out of 165 points are classified correctly for DMV-2, and 22 out of 24 points are correct for DMV-3. AdaBoost does well on DMV-3, with only 1 misprediction out of 24, but fails to learn the space for DMV-2: the algorithm only attempts to make a prediction 7% of the time, and is 75% accurate when it does predict. However, this failure to learn is apparent at the end of the training phase. The accuracy on the training set indicates that the classifier did not learn the space sufficiently well and therefore should not be used.

5.6 Comparison of AdaBoost and Random Decision Trees

It is apparent that Random Decision Trees perform better on this problem than our implementation of AdaBoost. In order to understand why, we analyze the bias and variance of both methods, using the following procedure [9]. We fix the test set, and generate 100 training sets of the same size by sampling with replacement from a much larger training set. We keep the size of the training set constant for this comparison, and focus on variance due to training set composition only.

Having generated 100 training sets and a single test set, we train each algorithm on each of the training sets, and evaluate the learned models against the test set. The *major prediction* is the label most commonly predicted by the 100 models. For example, if 50 models predict query plan 1, 30 models predict plan 2 and 20 models predict plan 3, then the major prediction is plan 1. Bias is the percentage of major predictions that are incorrect, i.e. do not match the true labels. In a test set of size 500, if 400 major prediction are correct, then bias is $1 - 0.8 = 0.2$.

Informally, variance quantifies the disagreement between models learned on different training sets. Consider a single point that represents an instance of the query template in the test set. If 50% of the models predict plan 1 for this point, 30% predict plan 2, and 20% predict plan 3, then the variance the algorithm on this point is the sample variance of predictions *away from the major prediction*, and is calculated as $\sqrt{\frac{(50-50)+(30-0)+(20-0)}{100}} = \sqrt{0.5}$. The variance of the algorithm on the test dataset is its average variance over all points in the dataset, 500 in our example.

AdaBoost sometimes returns an *uncertain* prediction. We have been treating such cases as a separate class when reporting on the accuracy of this algorithm. We take the same approach when evaluating bias and variance of AdaBoost in this section: uncertain predictions are considered neither correct nor incorrect, and are removed from the set of results before running the bias and variance analysis. In other words, we only quantify bias and variance of AdaBoost for the cases when the algorithm does make a prediction.

Table 18 presents the bias and variance of AdaBoost and Random Decision Trees on all datasets used in our experiments. Clearly, both bias and variance of RDT is consistently lower than that of AdaBoost, on every test run.

Our two algorithms use different methods to reduce variance: Random Decision Trees use randomization and averaging, while AdaBoost uses voting combined with weighted averaging. Consistently lower variance of RDT indicates that the combination of randomization in decisions tree construction and averaging posterior probabilities may be more effective than voting in our setting.

The difference in bias between the two methods comes primarily from the design of the weak learners in AdaBoost, compared to the construction of individual decision trees in RDT. The two algorithms are quite different, but intuitively, the difference boils down to the way an individual weak learner and an decision tree evaluate the probability of a particular plan, given a single predicate selectivity. An AdaBoost weak learner always splits the selectivity range into a fixed number of buckets, to ensure that probabilities across weak learners are comparable. RDT does not have this restriction, and is able to

			Bias		**Variance**	
Template	**Scale**	**Space**	**RDT**	**ADB**	**RDT**	**ADB**
TPCW-1	20	OPT	0.04	0.24	0.07	0.26
TPCW-1	100	OPT	0.05	0.18	0.12	0.26
TPCW-2	20	OPT	0.23	0.40	0.28	0.32
TPCW-2	100	OPT	0.28	0.35	0.29	0.33
TPCW-3	20	OPT	0.32	0.42	0.17	0.42
TPCW-3	100	OPT	0.20	0.22	0.16	0.23
DMV-1	60	OPT	0.14	0.23	0.08	0.25
TPCW-2	100	HYB	0.14	0.49	0.19	0.34
TPCW-3	20	HYB	0.02	0.28	0.11	0.35
TPCW-3	100	HYB	0.07	0.32	0.12	0.27
DMV-1	60	HYB	0.03	0.13	0.02	0.09

Fig. 18. Bias and variance of AdaBoost and RDT

generate a closer approximation of the true plan decision boundary. This flexibility of RDT makes the greatest difference at the boundary itself, where most mistakes are made. This is illustrated in Figure ?? for the query TPCW-1.

6 Discussion

As was demonstrated, machine learning techniques can achieve a significant overall performance improvement for some queries. However, these techniques are only practical if it is also possible to automatically identify cases where using ReoptSMART is *not* beneficial, and even impairs performance.

For AdaBoost, the accuracy on the training set is a good indication of accuracy on unseen data, provided that it comes from the same distribution as the training set. If the classifier fails to achieve the desired accuracy on the training set, or returns the "no plan" prediction too often (as for DMV-2), the system declares that the classifier cannot be used, and possibly collects more training data.

Random Decision Trees always achieve very high accuracy on the training dataset. However, this accuracy carries no guarantee that the algorithm will perform well on unseen data. A simple validation technique can be adapted to verify whether an RDT classifier is ready to be used for a particular query template. Similar to our experiments, the training dataset can be split into two subsets. RDT can be trained on the first subset, and validated against the second.

As an additional safety measure that ensures that the classifiers are up-to-date and accurate, the optimizer may be invoked to validate a number of recent predictions made by ReoptSMART. This is done when the system is idle. If the classifier no longer achieves the desired accuracy, it can be re-trained on this more recent data set.

ReoptSMART targets workloads of a large number of fast-executing parametric queries, with the objective of optimizing the aggregate behavior of the system. We thus target average-case performance in our implementation. An alternative, and possibly conflicting, objective, may be to target worst-case performance. Please see [21] for a discussion of risk vs. opportunity considerations. Our machine learning methods can be configured to optimize for different objectives in a simple yet principled way: by adjusting the loss function used during training. To optimize for the average case, we are penalizing all mis-predictions during the training phase equally. If our objective were to bound worst-case performance, our loss function would incorporate the actual mis-prediction penalty, possibly in an exponential way.

We report average-case results in our experimental evaluation. However, we also measured worst-case results. Our worst-case behavior is strictly better than OPT for most queries. For some worst-case queries, the cost of ReoptSMART is a significant multiple of the cost of OPT. These results, however, cannot be compared directly to the maximum degradation results reported in AniPQO[16]: we base our measurements on actual execution time, while AniPQO is based on optimizer estimates that are more likely to be stable at plan boundaries, where most mis-predictions take place. Our worst-case performance may also be biased by the fact that all our queries have very short execution times.

7 Related Work

Ioannidis and Kang [24] describe the applicability of several randomized algorithms to Parametric Query Optimization, applied to run-time parameters that take on discrete values, such as the number of buffer pages allocated to a query and types of available indexes. Our solution works for both discrete and continuous parameters.

Reddy and Haritsa [26] study properties of plan spaces on a suite of commercial query optimizers. The authors challenge the foundations underlying traditional geometric PQO approaches by observing that plan optimality regions may not be convex and may not be limited to a single contiguous region. We observed this for some of our query templates, e.g., DMV-1, and were able to learn such plan spaces successfully.

Hulgeri and Sudarshan [15,16] propose AniPQO: a geometric solution to PQO that samples the parameter space and approximates boundaries between regions of plan optimality by constructing an explicit decomposition of the space. AniPQO extends the optimizer to enable *plan-cost evaluation probes* that return the estimated cost of a given plan at a given point in the parameter space. Commercial optimizers do not readily provide such functionality. Hulgeri and Sudarshan argue that such probes are cheaper than regular optimization and extend a Volcano-based optimizer with which they are working to provide for such probes. In working with a commercial optimizer, we found that plan-cost evaluation probes would be nearly as expensive as regular optimizer calls. Considering that the probes also produce cost estimates, and not actual costs, we decided against this type of evaluation and evaluated our methods with respect to actual execution time. For this reason, we are unable to present a direct side-by-side comparison between AniPQO and ReoptSMART.

ReoptSMART learns the plan space using no more than 200 training points in all experiments, compared to over 1000 points for many queries in [16].

This advantage comes from variance-reducing techniques inherent to RDTs and AdaBoost. While geometric in nature, RDTs and AdaBoost are both less sensitive to the shape of the plan space compared to explicit geometric techniques like AniPQO. AniPQO was demonstrated to work on a class of SPJ queries with a "smooth" plan space: optimality regions are convex and there are no discontinuities in the plan space. We show here that ReoptSMART works for a larger class of queries: SPJ and aggregate, with continuous and discrete attributes, on uniform and skewed data sets.

More general kinds of templates have been given in the literature. In PLASTIC [13], a dynamic clustering algorithm is presented that groups queries that may have different query templates but for which the optimizer will generate the same execution plan template. Queries are clustered according to a similarity metric that includes both structural and statistical features. The clustering mechanism uses no variance-reducing techniques such as voting or averaging, and is sensitive to the variance due to training set composition.

A large body of work deals with query re-optimization at run-time. Antoshenkov [4] introduces the competition model of query execution: a query starts executing concurrently using different plans. Progress of the query along the different execution paths is monitored until it becomes apparent which plan is better than the others. However, redundant execution uses system resources unnecessarily.

Kabra and DeWitt [20] propose a mid-query re-optimization technique where run-time collection of statistics takes place at intermediate points in query execution. These statistics are compared against the optimizer's estimates, and if plan sub-optimality is detected, the query execution plan is changed, and the query is re-written to access materialized intermediate results.

In [21], Markl et al. extend the work of [20] and develop a progressive optimization framework that evaluates progress of query execution and may choose to re-optimize the query at a checkpoint. The authors provide an excellent overview of related work and study the risk vs. opportunity trade-off of their approach compared to others.

In Eddies [5], a query processing mechanism is introduced that continuously reorders operators in a query plan. This approach, while very flexible, is also non-trivial to implement and may result in non-negligible overhead at run-time.

Because the cost of testing is small, ReoptSMART could be used to optimize correlated subqueries. Queries containing correlated subqueries often cannot be rewritten into a single query block. In such cases, commercial optimizers optimize the outer query and the inner subquery as separate query blocks. From the point of view of the inner subquery, the outer references are parameters. (The inner query may also have bind variables of global scope.) The inner subquery will be called multiple times with various values for the parameters. To our knowledge, commercial systems currently optimize a query once. While some systems may abandon a query execution and reoptimize when the current execution looks bad [21], no system will routinely reoptimize a correlated subquery depending on the values of the outer reference.

There are good reasons not to invoke the optimizer for each outer reference. First, the optimization time may be significant, and would be multiplied by the number of outer references. Second, in order for an optimizer to generate a plan that calls for invocation of the optimizer itself at run-time, the optimizer needs to have a cost model for how long it thinks optimization would take [17]. These reservations do not apply, however, to a parametric query optimizer, as long as the execution time is small and can be accurately estimated. Using a single plan for a correlated subquery may be much worse than optimal. Using PQO, one could cheaply decide which plan to use for the correlated subquery, dynamically at run-time.

8 Conclusions and Future Work

We have demonstrated that machine learning methods can be used to accurately model predictions of a relational query optimizer. Based on these models, one can derive plans much more cheaply than the optimizer can. Further, the plans generated this way perform better than using a single pre-optimized plan for a query template.

Previous PQO methods only considered optimizer cost estimates and reasoned about the misprediction penalty in terms of those estimates. In our work we measure query execution time, and derive the actual misprediction penalty. We are the first to demonstrate that a PQO approach can result in savings beyond query optimization time, and achieve a significant overall net win over the methods currently available in relational database systems.

Since the optimizer provides just a model of the execution cost, sometimes the actual best plan is not the one chosen by the optimizer. We show how to "correct" the optimizer's plan selection within the machine learning model, using actual performance results for a query template. A performance improvement of more than an order of magnitude can be achieved for some queries.

In the future we would like to investigate incremental re-training opportunities for ReoptSMART, and to evaluate the performance of our method in presence of multidimensional selectivity estimation.

References

[1] Oracle database performance tuning guide 10g. http://download-east.oracle.com/docs/cd/B19306_01/server.102/ b14211/toc.htm.

[2] Stanford Racing. www.stanfordracing.org.

[3] TPC: Transaction processing performance council. www.tpc.org.

[4] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *ICDE*, 1993.

[5] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query optimization. In *SIGMOD*, 2000.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[7] G. Di Fabbrizio et al. AT&T help desk. In *ICSLP*, 2002.

[8] T. G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *CoRR*, cs.AI/9501101, 1995.

[9] P. Domingos. A unified bias-variance decomposition and its applications. In *ICML*, pages 231–238. Morgan Kaufmann, 2000.

[10] W. Fan et al. Effective estimation of posterior probabilities: Explaining the accuracy of randomized decision tree approaches. In *ICDM*, 2005.

[11] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. of Computer and System Sciences*, 55(1):119–139, 1997.

[12] Y. Freund and R. E. Schapire. A short introduction to boosting. *J. of Japanese Society for Artificial Intelligence*, 14(5):771–780, 1999.

[13] A. Ghosh et al. Plan selection based on query clustering. In *VLDB*, 2002.

[14] G. Graefe et al. MS SQL Server 7.0 query processor. www.microsoft.com/technet/prodtechnol/sql/70/maintain/sql7qp.mspx.

[15] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB*, 2002.

[16] A. Hulgeri and S. Sudarshan. AniPQO: Almost non-intrusive parameteric query optimization for nonlinear cost functions. In *VLDB*, 2003.

[17] I. F. Ilyas et al. Estimating compilation time of a query optimizer. In *SIGMOD*, 2003.

[18] Y. Ioannidis et al. Parametric query optimization. *VLDB J.*, 6(2):132–151, 1997.

[19] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD*, 1990.

[20] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[21] V. Markl et al. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[22] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[23] C. S. Mullins. Coding DB2 SQL for performance: The basics. www-106.ibm.com/developerworks/db2/library/techarticle/0210mullins/0210mullins.html.

[24] V. Poosala and Y. Ioannidis. Selectivity estimateion without the attribute value independence assuption. In *VLDB*, 1997.

[25] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[26] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, 2005.

[27] R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[28] K. Tumer and J. Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(3-4):385–403, 1996.