F O U N D A T I O N S O F C O M P U T I N G A N D D E C I S I O N S C I E N C E S
Vol. 49 (2024) No. 2
ISSN 0867-6356
DOI: 10.2478/fcds-2024-0011 e-ISSN 2300-3405

Learning From User-Specified Optimizer Hints in Database Systems

Maciej Zakrzewicz *

Abstract. Recently, numerous machine learning (ML) techniques have been applied to address database performance management problems, including cardinality estimation, cost modeling, optimal join order prediction, hint generation, etc. In this paper, we focus on query optimizer hints employed by users in their queries in order to mask some Query Optimizer deficiencies. We treat the query optimizer hints, bound to previous queries, as significant additional query metadata and learn to automatically predict which new queries will pose similar performance challenges and should therefore also be supported by query optimizer hints. To validate our approach, we have performed a number of experiments using real-life SQL workloads and we achieved promising results.

Keywords: database systems, query optimization.

1. Introduction

Database query optimizer hints are commonly used by query developers to override the default Query Optimizer behavior. The hints are considered "patches" to fix wrong decisions about the optimal query execution plan selection. Such wrong decisions usually result from imprecise cardinality estimates, missing or stale optimizer statistics, suboptimal join orders, ineffective query transformations, etc. Although developers are instructed to use hints sparingly, the mechanism often becomes the last resort, when query performance is far from satisfactory.

There are various implementations of optimizer hints in modern RDBMSs and NoSQL database servers. PostgreSQL allows developers to narrow execution plan search space by setting session-level parameters to allow or disallow: indexes, specific types of join algorithms, specific join orders, etc. In Oracle Database, optimizer hints can be specified in the

* Institute of Computing Science, Poznan University of Technology, Poznan, Poland.

form of specially formatted comments, embedded in SQL queries (see Figure 1). They allow for more precise control over query execution plans by referring to specific tables and specific indexes of the query. Oracle Databases also provide externalized hint sets, called SQL Plan Baselines, which are applied automatically when identical SQL queries are issued again. Microsoft SQL Server provides dedicated syntax of hint clauses to be included as parts of SQL queries. MySQL/MariaDB support both comment-based hints and hint clauses. Similar mechanisms are available in some NoSQL systems, like MongoDB.

```
SELECT /*+ USE_NL(emps dept) INDEX(emps emp_city_idx) */
last_name, dept_name
FROM emps JOIN dept ON (emps.dept_no = dept.dept_no)
WHERE city_name = 'Warsaw'
```

Figure 1. Sample hinted SQL query (Oracle Database): forcing the Nested Loops join algorithm and the use of emp_city_idx index

In this paper, the presence of optimizer hints in a query will be considered an evidence (flag) that otherwise the Query Optimizer would choose a bad query execution plan. Developers do not use optimizer hints without a purpose. As it is quite time consuming to figure out a good, working set of optimizer hints, the developers use this tool only when no other query tuning techniques have led to acceptable query response times. Therefore, the queries that use optimizer hints should be interpreted as the queries for which the Query Optimizer failed to properly estimate costs of its execution plans in order to pick the optimal one.

If such "problematic" queries were identified in advance, the Query Optimizer would be able to engage extra tuning techniques to avoid the performance problems - like empirical testing of alternative execution plans (e.g. Oracle SQL Tuning Advisor) or Dynamic Sampling used by Oracle Database for cardinality estimation when the optimizer statistics fail to properly describe data distribution. Since such extra tuning techniques tend to be costly, they cannot be applied to all queries. Being able to guess, which of the arriving queries should attract attention, would support the efficient use of the advanced optimization mechanisms.

We argue that optimizer hints can be considered additional query metadata/flags (expertise) and can be used to train ML models that proactively help recognize queries which will pose performance-related risks. The ML models may use automatically collected SQL query texts to inherit relationships between the query texts and the needs to correct the Query Optimizer behavior. When the models are applied on new queries and the need for optimizer hints is predicted, two resulting actions can be taken: (1) the Query Optimizer can be simply "warned" of potential performance-related problems and can use its advanced optimization mechanisms, or (2) the predicted optimizer hints can be even applied automatically to the new queries to force the Query Optimizer to behave adequately. We believe this approach can be especially effective in Data Warehouse environments, where reporting queries are not trivial and their response time expectations are significant.

The key contributions of this paper are: (1) we propose a new application of Machine Learning methods in the area of database systems and demonstrate its usability, (2) we introduce a novel framework to use query optimizer hints metadata related to the past queries in order to predict which future queries will suffer from undesirable default behavior of the Query Optimizer, (3) we demonstrate experimental results showing the accuracy and

efficiency of the framework and the success of the concept. We are not aware of any existing work on Machine Learning based on query optimizer hints.

The remainder of this paper is organized as follows. We discuss related work in Section 2. Section 3 describes the architecture and features of the framework. Section 4 contains experimental results. In Section 5 we present our conclusions and comment on open problems.

2. Related Work

Database Servers enable developers to influence the process of selecting the optimal query execution plan. PostgreSQL [22] provides a number of session-level and server-level parameters that narrow the search space of query execution plans. Oracle Database [20] additionally supports Optimizer Hints included in SQL query text and offers adaptive query optimization mechanisms, e.g. in the form of SQL Plan Directives to override inaccurate statistics, or SQL Plan Baselines/SQL Profiles to freeze execution plans for a query. Similar tools are provided by Microsoft SQL Server [18], MySQL [19] and several NoSQL DBMSs.

Machine Learning has been recently considered a promising technique for database query optimization, e.g. Leo [24], which adjusted histogram estimators by monitoring similar queries, [10][15][25][28] proposed to use deep learning to learn cardinality estimations or query costs, in [1][2][3] cardinality estimation is based on query driven approaches, [29] [30] described unsupervised Monte Carlo-based solutions, in [8] cardinalities were estimated via query containment rates, [12][17] demonstrated that reinforcement learning helps find good query execution plans. In [10] the authors proposed to use a multi-set convolutional network to predict join-crossing correlations in the data. Database queries were represented as a collection of: a set of tables, a set of joins and a set of predicates. In [25] an end-to-end learning-based cost estimation framework using a tree-structured deep neural network model was described, able to estimate both database query cost and cardinality simultaneously. A database query was encoded/modeled as: query operation, meta data, query predicate and some data samples. The learned model contained embedding layer, representation layer and estimation layer. [28] used multi-set convolutional neural networks for cardinality estimation. The main assumption was that a sufficiently deep neural network can model very complex data dependencies and correlations. The authors build different neural networks for various subparts of the database schema instead of having one global neural network for the whole schema at once. In [1], the authors focused on the Set Cardinality Prediction (SCP) problem for distance-based nearest neighbors (dNN) queries. They applied unsupervised competitive learning and hetero-associative competitive learning models to use previous dNN queries in order to learn the answers to incoming dNN queries. [2] described an Unsupervised Competitive Learning that predicted the cardinality of an unseen query without requesting its execution and it was able to adapt to new query patterns and data updates. [3] proposed the use of a unsupervised regression function estimation model to estimate the cardinality (number of vectors) of multidimensional data subspaces, defined by query selections over multi-dimensional datasets, for two types of database queries: range queries and distance nearest neighbors (radius) queries. In [30], the authors demonstrated that selectivity estimation could be performed with high accuracy by

using deep autoregressive models and in [29], a join cardinality estimator called NeuroCard was presented, being a single neural density estimator over an entire database. It was able to capture correlations across multiple joins in a single deep autoregressive model, without any independence assumptions. Once trained, it could handle all queries issuable to the schema, regardless of what subset of tables was involved.

Research papers on applying reinforcement learning to adaptive query processing [26] [27] showed interesting results. [26] described a framework, called SkinnerDB, which used reinforcement learning (Bandit-based Monte-Carlo planning) to learn optimal join orders on the fly, during the execution of the current query. It required no data statistics, no costs or cardinality models. [27] focused on adaptive query processing, where query plans are improved at runtime by means of feedback, treating query execution as a process of routing tuples to the query operators that combine to compute a query. They proposed a general framework for the routing problem, leveraging reinforcement learning.

Machine learning techniques have been also commonly applied in the area of SQL Injection Attack detection. [6] studied how Naive Bayes, Random Forest, SVM, ID3, and K-means could help detect unknown SQL Injection statements that had not been previously used in the training data. Query features were chosen based on access behavior mining and a grammar pattern recognizer. The author of [31] applied Random Forest, Logistic Regression, SVM, Multilayer Perceptron (MLP), Long Short-Term Memory (LSTM), and a Convolutional Neural Network (CNN) to detect SQL Injection vulnerabilities in PHP code. It was reported that CNN provided the best precision. [13] proposed an adaptive deep forest model (ADF) with the integration of the AdaBoost algorithm to predict SQL Injection Attacks. The achieved efficiency was comparable to that of traditional machine learning models, such as decision trees, and better compared with regular deep neural network models, such as RNN and CNN. [14] described an SQL Injection Attack detection framework based on Support Vector Machines (SVM) and an improved Term Frequency Inverse Document Frequency (TF-IDF) algorithm. In [21], the authors studied how data preparation and feature extraction techniques based on TF-IDF influenced detection accuracy of malicious SQL queries. [16] described the use of TF-IDF-CHI algorithm in SQL Injection Attack detection.

Machine learning algorithms were also applied to database index structures [11], index selection [4], query latency prediction [5].

3. System Model

The architecture of our framework is shown in Figure 2. In the first phase, we capture all SQL queries together with their optimizer hints (if any). The captured SQL queries are separated from their optimizer hints and stored in the Workload Repository. Next, a periodic offline task (e.g. monthly) transforms each collected SQL query and its optimizer hints (if any) into vectorized features. Then, a ML model (ensemble of Random Forests) is trained to be able to predict optimizer hints for new SQL queries. When the training has been completed, the ML model is being used in the Query Optimizer loop to help detect SQL queries which need special treatment. For the queries that the ML model recommends to add optimizer hints to, we add the hints automatically, while optionally we can also mark the queries to participate in extra, extensive automatic SQL tuning tasks (e.g. automatic

SQL Tuning Advisor of Oracle Database). The whole process is periodically reiterated on new workload.

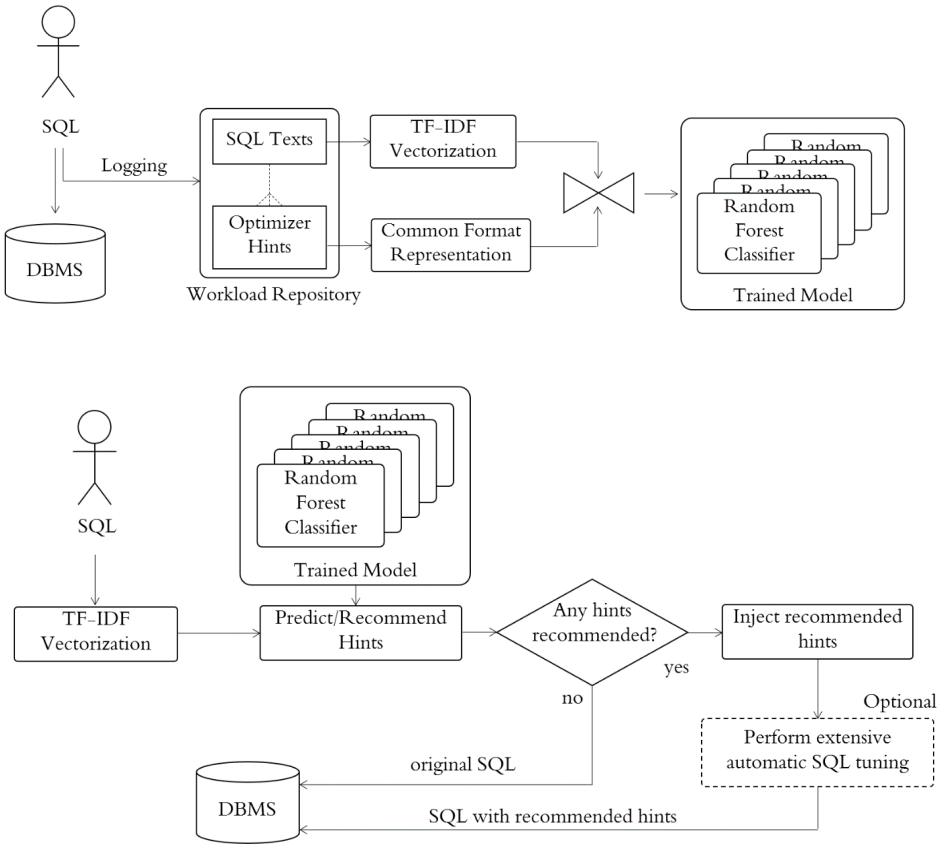

Figure 2. Framework architecture: training subsystem and recommendation subsystem

3.1. Optimizer Hints Modeling

Due to different syntactic formats employed by DBMSs, we introduced the Common Hint Format (CHF), which allows us to universally express optimizer hints by using JSON documents. CHF descriptors may contain four different types of hints, associated with query blocks: (1) binary hints (e.g. *enable_seqscan* in PostgreSQL to allow/disallow table sequential scans), (2) ordered hints (e.g. /*+ *LEADING* */ in Oracle Database to force specific join order), (3) categorical hints (e.g. *FORCE INDEX* in MySQL to force the specified index),

(4) quantitative hints (e.g. */*+ *CARDINALITY* */* in Oracle Database to advise the Query Optimizer about the expected cardinality). Table 1 shows examples of hinted SQL queries and their corresponding CHF descriptors for different types of DBMSs. Generating a CHF descriptor for a SQL query is straightforward.

Table 1. Examples of Common Hint Format representation of real-life optimizer hints

`-- ORACLE`
`SELECT /*+`
`  LEADING(e2 e1)`
`  USE_NL(e1)`
`  INDEX(e1 emp_id_pk)`
`  USE_MERGE(j)`
`  FULL(j)`
`  NO_REWRITE`
`  CARDINALITY(j,10) */`
`...`
	`{ optimizer_hints: [`
	`  {query_block: "main",`
	`  hints: {`
	`  cardinality_j_10: 1,`
	`  no_rewrite: 1,`
	`  full_j: 1,`
	`  use_merge_j: 1,`
	`  index_e1_emp_id_pk: 1,`
	`  use_nl_e1: 1`
	`  leading_e2_e1: 1`
	`  }`
	`  }] }`

`-- POSTGRESQL`
`SET ENABLE_SEQSCAN=OFF;`
`SET ENABLE_NESTLOOP=OFF;`
`SET ENABLE_MATERIAL=OFF;`
`SELECT ...`
	`{ optimizer_hints: [`
	`  {query_block: "main",`
	`  hints: {`
	`  not_enable_seqscan: 1,`
	`  not_enable_nestloop: 1,`
	`  not_enable_material: 1`
	`  }`
	`  }] }`

`-- MYSQL`
`SELECT /*+ JOIN_ORDER(t1, t2) */ *`
`FROM t2 NATURAL JOIN t1`
`  IGNORE INDEX (col3_idx)`
`WHERE ...`
	`{ optimizer_hints: [`
	`  {query_block: "main",`
	`  hints: {`
	`  ignore_index_col3_idx: 1,`
	`  join_order_t1_t2: 1`
	`  }`
	`  }] }`

`-- MONGODB`
`db.users.find().hint("age1")`
	`{ optimizer_hints: [`
	`  {query_block: "main",`
	`  hints: {`
	`  index_age1: 1`
	`  }`
	`  }] }`

3.2. Feature Extraction

First, a form of canonicalization is performed on the SQL query texts. All comments and optimizer hints are removed from SQL query texts. Table aliases, column aliases, whitespaces inside function references, whitespaces around operators and stopwords like

"ON", "USING", "AND", "OR", "NOT", "FROM", "SELECT", "WHERE" are also removed (eg. "FROM invoices inv" becomes "FROM invoices", "ROUND(␣price␣/␣10␣)" becomes "ROUND(price/10), "WHERE x␣=␣y" becomes "WHERE x=y"). All characters are converted to upper case and all new line characters are converted to white spaces. All these canonicalization steps help us remove meaningless terms as well as combine pieces of important query expressions into single terms.

Next, for SQL query text vectorization, we use the popular Term Frequency - Inverse Document Frequency (TF-IDF) method, considering SQL queries as sentences of words separated by whitespaces. TF-IDF has been commonly considered as SQL query text vectorization method [14][16][21]. Although SQL queries seem to be structural, there are no obvious syntactical markers of their performance-related properties, therefore, we decided to use this vectorization technique, known from the field of natural language processing. The TF-IDF consists of two phases. In the first step, TF estimates the importance of a term in a query as the number of its occurrences divided by the number of all terms in the query. In the second step, IDF is calculated as logarithm of the total number of queries in the set divided by the number of queries where the term appears. Terms unique to a small percentage of SQL queries receive higher importance values than words that are common across all queries. Finally, the TFs are multiplied by IDFs to get the TF-IDF importance of each term and the queries become multi-attribute vectors, with attributes representing the terms and TF-IDFs being their values. The below example shows how a sample 2-query set gets vectorized using the TF-IDF method.

Example.
Consider the following sample query set (hints removed):

```
query1: "SELECT last_name, dept_name FROM emps
         JOIN dept ON (emps.dept_no = dept.dept_no) WHERE id = ?",
query2: "SELECT last_name FROM emps WHERE id = ?"
```

Canonicalization will transform the queries in the following way:

```
query1: "LAST_NAME, DEPT_NAME EMPS
         JOIN DEPT (EMPS.DEPT_NO=DEPT.DEPT_NO) ID=?",
query2: "LAST_NAME EMPS ID=?"
```

TF values for the sample queries' terms are shown below:

	DEPT	DEPT_NAME	EMPS	(EMPS...DEPT_NO)	ID	JOIN	LAST_NAME
query1	1/7	1/7	1/7	1/7	1/7	1/7	1/7
query2	0	0	1/3	0	1/3	0	1/3

IDF values for the sample queries' terms are shown below:

DEPT	DEPT_NAME	EMPS	(EMPS...DEPT_NO)	ID	JOIN	LAST_NAME
log(2/1)	log(2/1)	log(2/2)	log(2/2)	log(2/2)	log(2/1)	log(2/2)

Thus, the vectorized query set will be represented as follows:

	DEPT	DEPT_NAME	EMPS	(EMPS...DEPT_NO)	ID	JOIN	LAST_NAME
query1	0.043	0.043	0.000	0.043	0.000	0.043	0.000

```
query2  0.000     0.000  0.000            0.000 0.000  0.000     0.000
```

3.3. Classification

The task of predicting optimizer hints is essentially a multi-label classification problem, with multiple nonexclusive labels (optimizer hints) assigned to data instances (vectorized SQL queries). Vectorized training query sets are expected to have hundreds of attributes (terms), while expected numbers of distinct optimizer hints used seem to be in the range of 10-20. In real-life scenarios, we gather hundreds or thousands of distinct SQL queries per a DBMS, while hinted queries are quite uncommon, occurring in roughly 10% of all queries (which may require imbalance compensation by e.g. down-sampling non-hinted queries).

We followed a popular approach of Binary Relevance [23], independently training one binary classifier for each label. All distinct optimizer hints occurring in the training query set formed the set of labels and then Random Forest classifiers were trained to predict the need for each of the possible optimizer hints in a SQL query. The choice of the Random Forest classifier [9] was justified by high dimensionality of the vectorized training sets - Random Forest can effectively deal with a large number of input variables without feature selection or dimensionality reduction techniques. The number of trained classifiers equals the number of distinct optimizer hints observed in the training query set. In general, query optimizer hints are independent on each other with very rare exceptions e.g. when the use of a specific join method should be accompanied by the use of a specific index. In our approach we decided to ignore such potential dependencies.

3.4. Generating Hint Recommendations

Given a new, unseen SQL query, we first use the same TF-IDF-based vectorization as in the training phase to convert the SQL query into a multi-attribute vector. Then, we execute all our Random Forest classifiers to predict whether their specific optimizer hints should be applied to the query. Based on the positive predictions, a set of recommended optimizer hints is created.

4. Experiments

We evaluated the effectiveness of using optimizer hints metadata related to the captured queries in order to predict which future queries will suffer from undesirable default behavior of the Query Optimizer. We observed the quality of the trained models by verifying whether: (1) the queries required **any** type of optimizer hints to perform optimally (a "warning" to the Query Optimizer), (2) the queries required our **recommended** optimizer hints to perform optimally (to "fix" the queries automatically). As there exist no official benchmarks for this type of testing, we used three sets of real-life, proprietary reporting SQL queries recorded in three different data warehouse/business reporting environments, based

on PostgreSQL 15 and Oracle Database 12c/19c DBMSs. In PostgreSQL, optimizer hints were recorded using the native logging collector. In Oracle Database, we used SQL Trace, SQL Plan Baselines and SQL Plan Directives to grasp all optimizer hint sources. The details of the query sets are given in Table 2: the number of database tables, the number of recorded queries, the number of queries that originally contained any optimizer hints, the average number of hints per query in hinted queries, the average number of tables per query. We wish we had larger query sets for training the models, however, this is actually a real-life scenario we should expect.

Table 2. SQL workload query sets characteristics

set id	DBMS	#tables	# queries	# queries with hints	avg #hints per query	avg #tables per query
A	Oracle	72	268	93	3.2	5.5
B	Oracle	47	191	22	2.7	7.2
C	PostgreSQL	17	80	19	2.6	6.0

The following optimizer hints were the most common in Set A and Set B: *INDEX* (to force a specific index), *NO_INDEX* (to block a specific index), *LEADING* (to force a specific join order), *REWRITE* (to force materialized view query rewrite), *PARALLEL* (to control parallel query execution), *FIRST_ROWS* (to prefer low response time over high throughput), *DYNAMIC_SAMPLING* (to not trust the statistics). In the Set C, the most frequent optimizer hints were: *enable_seqscan* (to block full table scans), *enable_nestloop* (to block the Nested Loops join algorithm), *enable_sort* (to force using indexes for sorting).

In order to compensate for imbalanced data (hints were present in only 11-34% of our SQL queries), we used down-sampling on the queries which contained no optimizer hints. The query sets were then randomly split 75:25 into training datasets and testing datasets, repeatably 10 times for each experiment to provide for stable estimators of evaluation measures. After ML models have been trained with the training datasets, their performance was tested using the testing datasets. The testing process was split into the following phases: (1) for each SQL query with hints removed, the ML model predicted optimizer hints that should be applied to the query (if any), (2) two versions of each SQL query, a no-hints query and the query with recommended hints, were executed to measure their execution times, (3) for Oracle Database use cases only: on SQL queries, for which any hints were predicted, we ran the automatic SQL Tuning Advisor in extensive empirical mode to verify if any other hint-related performance improvements were possible beyond our predicted hints. The experiments were conducted with Weka [7], using the following Random Forest classifier parameters: number of iterations - 100, number of randomly chosen attributes - logarithm (number of inputs + 1), maximum depth of the tree - unlimited.

The experimental observations were categorized into five groups. The group "True positives (marked)" represents cases where the framework has **correctly** recommended that the query **would** perform better if **some** hints were added. "True positives (optimal)" (Oracle Database only) are cases where the framework has **correctly** recommended the **optimal** set of hints for the query to perform best (i.e. no automatic SQL tuning tool was able to provide a better query execution plan). The group "False positives (marked)" are the cases where the framework has recommended that the query would perform better if hints were

added, but the query **already** appeared **optimal** (no improvement). "True negatives" occurred when the framework **correctly** did not recommend to add any hints to the query and **in fact** the query appeared optimal, while "False negatives" represent cases where the framework did not recommend to add any hints to the query while in fact hints **could** be added to improve the query performance. The measured values (rounded averages from 10 experiments with 75:25 random splits) for the five groups of cases in each of the three sets of the test queries are shown in Table 3. The average true positive rate (sensitivity) for all test query sets (*TP/(TP+FN)*) was 83% for "marked" and 76% for "optimal", and the average true negative rate (specificity) for all test query sets (*TN/(TN+FP)*) was 82%. We also observed sensitivity and specificity differences between PostgreSQL and Oracle Database cases: for the PostgreSQL test queries we achieved the true positive rates of 81%, while for Oracle Database test queries the results were 83% ("marked") and 74% ("optimal"). The true negative rate obtained on the PostgreSQL test query set was 100%, while for Oracle Database test queries it was 79%.

Table 3. Confusion matrices for the test query sets

set id: **A** #queries: **67**		Actual values	
		Positive (hints needed)	Negative (hints not needed)
Predicted values	Positive (hints needed)	**37 marked** **22 optimal**	**7**
	Negative (hints not needed)	**16**	**7**

set id: **B** #queries: **48**		Actual values	
		Positive (hints needed)	Negative (hints not needed)
Predicted values	Positive (hints needed)	**11 marked** **7 optimal**	**5**
	Negative (hints not needed)	**29**	**3**

set id: **C** #queries: **20**		Actual values	
		Positive (hints needed)	Negative (hints not needed)
Predicted values	Positive (hints needed)	**9 marked**	**0**
	Negative (hints not needed)	**9**	**2**

In Figures 3, 4, 5 we give an overview of individual test query performance results. For only the first 20 queries from a single iteration of each test query set, the chart demonstrates

the original query execution times (without any optimizer hints), execution times observed when our model-recommended hints were applied, and optimal execution times, achieved after additional extensive tuning was performed. Additionally, in Figures 6, 7, 8, we illustrate execution time improvement for the queries that used the recommended hints (original query execution time minus hinted query execution time). For test query set A, in 7% of cases, it happened that our automatically recommended hints worsened the query execution times. Figure 9 shows the total performance improvement achieved on the three test query sets after applying all the recommended optimizer hints: set A - 3,289 s (vs. 5,620 s), set B - 4,000 s (vs. 5,396 s), set C - 1,270 s (vs. 1,521 s).

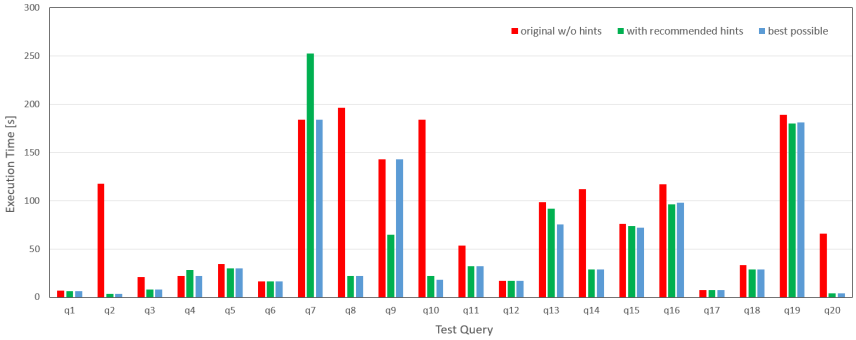

Figure 3. Sample test query performance: original (w/o hints) vs. recommended hints vs. best possible hints (Oracle Database Set A)

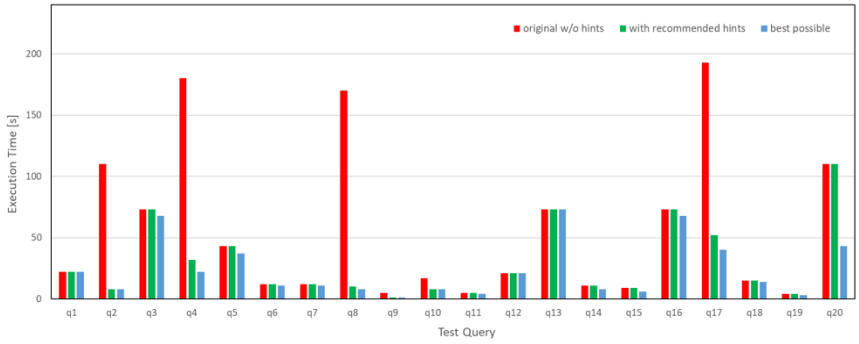

Figure 4. Sample test query performance: original (w/o hints) vs. recommended hints vs. best possible hints (Oracle Database Set B)

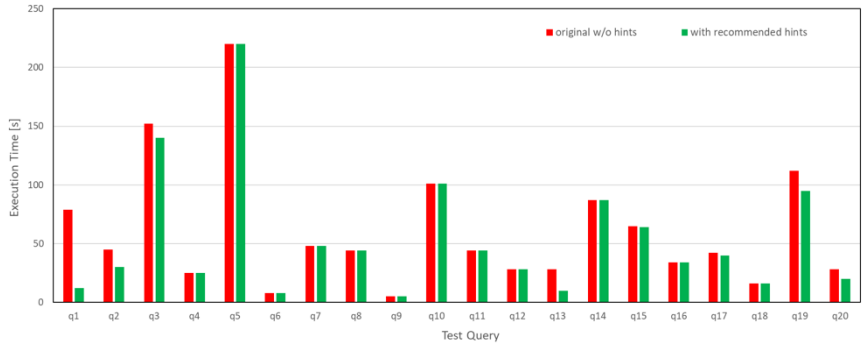

Figure 5. Sample test query performance: original (w/o hints) vs. recommended hints vs. best possible hints (PostgreSQL Set C)

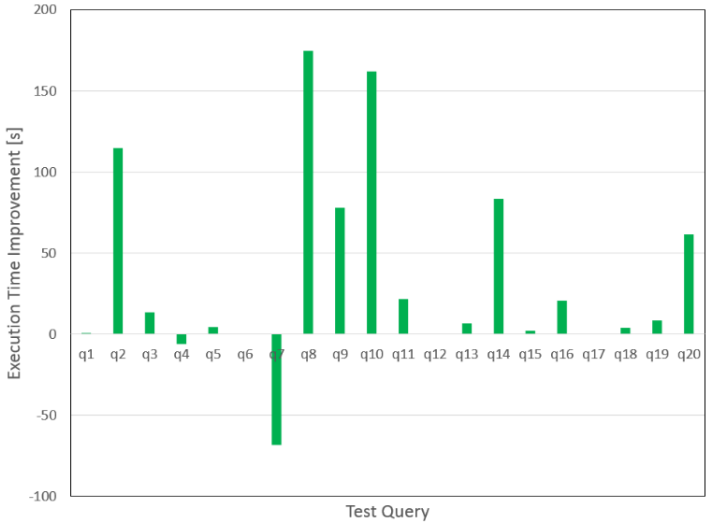

Figure 6. Execution time improvement achieved for sample test queries (Oracle Database Set A)

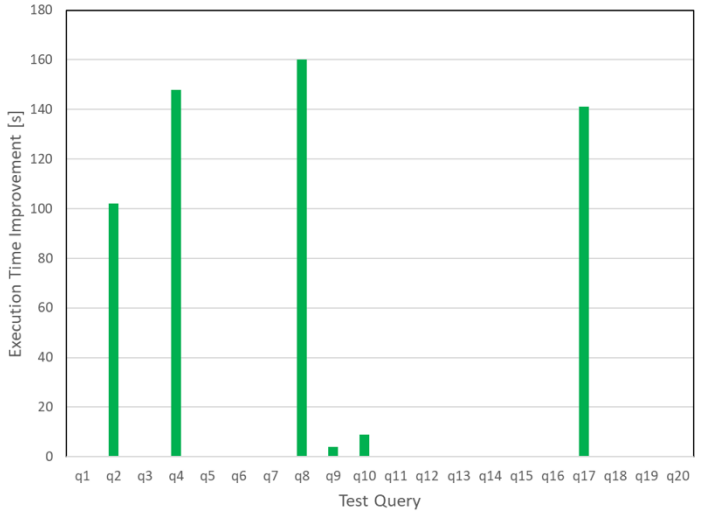

Figure 7. Execution time improvement achieved for sample test queries (Oracle Database Set B)

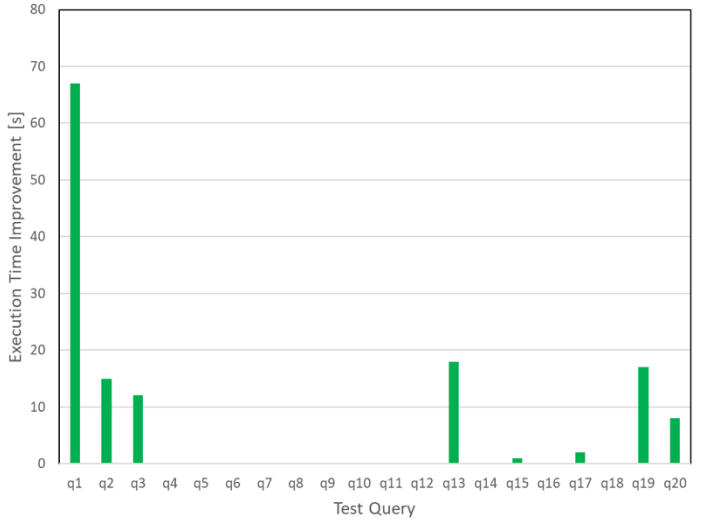

Figure 8. Execution time improvement achieved for sample test queries (PostgreSQL Set C)

Training times were negligible as our training sets were relatively small. Although in practical applications we would possibly be able to collect thousands of SQL queries in general, the distribution of un-hinted vs. hinted ones is likely to be highly unbalanced. We do not expect developers to put optimizer hints into every second query they run. Therefore, we do not consider the training time a significant factor.

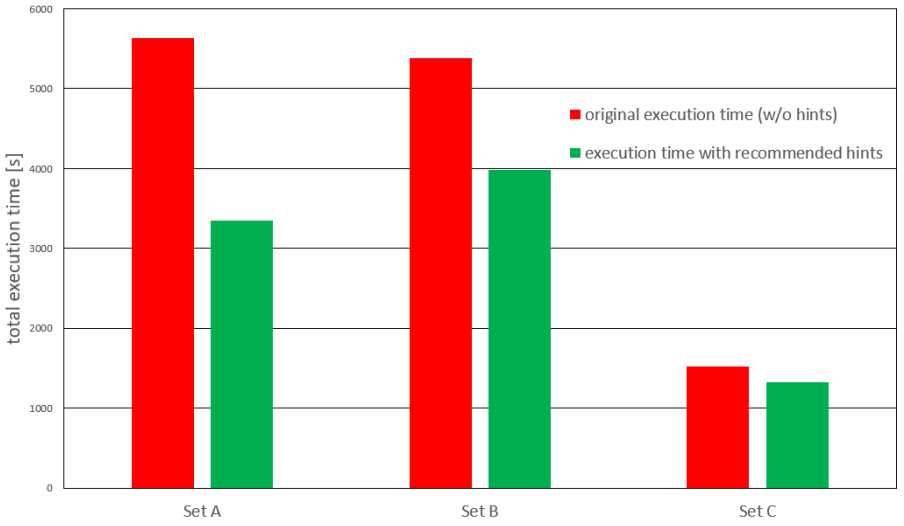

Figure 9. Total execution times for test query sets A, B, C: original vs. recommended hints

5. Conclusions

Our experiments provided us with very promising results although we were aware of the small size of the available training data sets. For a vast majority of the tested queries, the framework correctly marked the queries as requiring performance fine-tuning or as already optimal. The overall database performance (represented by the total execution times of the test query sets) was significantly improved. We demonstrated that user-specified optimizer hints are interesting indicators of expected query optimization problems and by observing the characteristics of database queries involving optimizer hints, we can learn how to recognize which of the new database queries will pose performance-related challenges to the Query Optimizer. Being able to automatically recommend optimizer hints for the new queries, we can either apply the recommended hints or just use the fact that there are any recommendations as an indicator to be more careful when optimizing the query. The experiments performed on real-life SQL query tests showed that trained ML models can accurately detect database queries that will become inefficient if no tuning steps are taken. On the other hand, although the automatically recommended optimizer hints where indeed advantageous in most cases, they might occasionally degrade the performance.

The proposed framework can be integrated with any DBMS, which supports optimizer hints. However, the integration can follow different directions. Continuous query monitoring would allow to recommend optimizer hints for some of the queries. In order to avoid eventual performance degradation due to possible bad hinting, two versions of such queries - the original one and the one with recommended hints - should be speculatively executed in parallel. When one query version completes, the other would be cancelled. Generally, machine learning methods applied to query optimization commonly suffer from "tail catas-

trophes", meaning they are better than traditional query optimizers on average, but they occasionally happen to perform catastrophically in the tail.

Another idea is to use automatic hint recommendations to only advise the user that potential performance improvement may be achieved, e.g. by displaying additional messages as part of EXPLAIN results or GUI tool notifications. The user can then decide whether or not to use the recommended hints. Yet another integration direction involves proactively advising the DBMS that query data sources may suffer from bad cardinality estimates (due to table data distribution, correlated columns, predicate expressions, etc.), leading to inefficient query execution plans. The optimizer hints usually just mask underlying problems, so detecting possible needs to use hints can clearly indicate such a problem in advance.

There is still a number of open issues to explore. One of the most important ones are query regressions. Since database/data warehouse content is not static, some of previously hinted queries may either no longer require the hints or can even suffer from suboptimal hinting. Therefore, by learning from such inadequately hinted queries, we would end up with unnecessary or suboptimal recommended hints. One of possible solutions is to retrain the ML model periodically on current workloads, believing that users would identify and fix their wrongly hinted queries by themselves. Another problem is related to performance of the recommendations generation algorithm. Although applying the ML model on the new data is relatively fast, the whole chain of the necessary processing steps, including the feature extraction, can have significant impact on the total query processing time. It does not seem to be a problem for a DSS system, where queries are usually complex and time consuming, but for OLTP systems the overhead might not be acceptable.

References

[1] Anagnostopoulos C., Triantafillou P., Learning Set Cardinality in Distance Nearest Neighbours. In *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM)*, ICDM ’15, pages 691–696, USA, Nov. 2015. IEEE Computer Society.

[2] Anagnostopoulos C., Triantafillou P., Learning to accurately COUNT with query-driven predictive analytics. In *2015 IEEE International Conference on Big Data (Big Data)*, Big Data ’15, pages 14–23, Oct. 2015.

[3] Anagnostopoulos C., Triantafillou P., Query-Driven Learning for Predictive Analytics of Data Subspace Cardinality. *ACM Trans. Knowl. Discov. Data*, 11(4):47:1–47:46, June 2017.

[4] Ding B., Das S., Marcus R., Wu W., Chaudhuri S., Narasayya V. R., AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *38th ACM Special Interest Group in Data Management*, SIGMOD ’19, 2019.

[5] Duggan J., Papaemmanouil O., Cetintemel U., Upfal E., Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT ’14, pages 109–120, 2014.

[6] Gao H., Zhu J., Liu L., Xu J., Wu Y., Liu A., *Detecting SQL injection attacks using grammar pattern recognition and access behavior mining*. In: 2019 IEEE International Conference on Energy Internet (ICEI). IEEE, 2019. p. 493-498.

[7] Hall M., Eibe F., Holmes G., Pfahringer B., Reutemann P., Witten I., *The WEKA Data Mining Software: An Update*. SIGKDD Explorations, 11.1, 2009, 10–18.

[8] Hayek R., Shmueli O., *Improved Cardinality Estimation by Learning Queries Containment Rates*. arXiv:1908.07723 [cs], Aug. 2019.

[9] Ho T.K., *Random decision forests*. In Proceedings of 3rd international conference on document analysis and recognition, 1995. pp. 278–282.

[10] Kipf A., Kipf T., Radke B., Leis V., Boncz P., Kemper A., Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[11] Kraska T., Beutel A., Chi E. H., Dean J., Polyzotis N., The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, New York, NY, USA, 2018. ACM.

[12] Krishnan S., Yang Z., Goldberg K., Hellerstein J., Stoica I., *Learning to Optimize Join Queries With Deep Reinforcement Learning*. arXiv:1808.03196, Aug. 2018.

[13] Li Q., Li W., Wang J., Cheng M., *A SQL injection detection method based on adaptive deep forest*. IEEE Access, 2019, 7: 145385-145394.

[14] Li Y., Bin Z., *Detection of SQL Injection Attacks Based on Improved TFIDF Algorithm*. Journal of Physics: Conference Series. 1395. 012013. 10.1088/1742-6596/1395/1/012013, 2019.

[15] Liu H., Xu M., Yu Z., Corvinelli V., Zuzarte C., Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[16] Liu R., Zhang W., *A detection methodology for SQL injection attacks based on the TF-IDF-CHI algorithm.* Proc. SPIE 12941, International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2023), 129410N https://doi.org/10.1117/12.3011777

[17] Marcus R., Papaemmanouil O., Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM @ SIGMOD '18, Houston, TX, 2018.

[18] *MS SQL Server Hints*, https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql

[19] *MySQL Optimizer Hints*, https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html

[20] *Oracle Database Influencing the Optimizer*, https://docs.oracle.com/en/database-/oracle/oracle-database/21/tgsql/influencing-the-optimizer.html

[21] Oudah M.A., Marhusin M.F., Narzullaev A., *SQL Injection Detection Using Machine Learning with Different TF-IDF Feature Extraction Approaches*. In: Al-Emran, M., Al-Sharafi, M.A., Shaalan, K. (eds) International Conference on Information Systems and Intelligent Applications. ICISIA 2022. Lecture Notes in Networks and Systems, vol 550. Springer, Cham. https://doi.org/10.1007/978-3-031-16865-9_57

[22] *PostgreSQL Query Planning*, https://www.postgresql.org/docs/current/runtime-config-query.html

[23] Read J., Pfahringer B., Holmes G., Frank E., Classifier Chains for Multi-label Classification. *Machine Learning Journal*. Springer. Vol. 85(3), (2011)

[24] Stillger M., Lohman G. M., Markl V., Kandil M., LEO - DB2's Learning Optimizer. In *VLDB, VLDB '01*, pages 19–28, 2001.

[25] Sun J., Li G., An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 13(3):307–319, Nov. 2019.

[26] Trummer I., Moseley S., Maram D., Jo S., Antonakakis J.. SkinnerDB: Regretbounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11(12):2074–2077, 2018.

[27] Tzoumas K., Sellis T., Jensen C., A Reinforcement Learning Approach for Adaptive Query Processing. *A DB Technical Report*, June 2008.

[28] Woltmann L., Hartmann C., Thiele M., Habich D., Lehner W., Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, pages 1–8, Amsterdam, Netherlands, July 2019. Association for Computing Machinery.

[29] Yang Z., Kamsetty A., Luan S., Liang E., Duan Y., Chen X., Stoica I., *NeuroCard: One Cardinality Estimator for All Tables*. arXiv:2006.08109, June 2020.

[30] Yang Z., Liang E., Kamsetty A., Wu C., Duan Y., Chen X., Abbeel P., Hellerstein J. M., Krishnan S., Stoica I., Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment*, 13(3):279–292, Nov. 2019.

[31] Zhang K., A machine learning based approach to identify SQL injection vulnerabilities. In: *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019. p. 1286-1288.

Received 19.10.2023, Accepted 15.02.2024