GALATASARAY UNIVERSITY

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

LOAD RELATED FEATURE ENGINEERING FOR QUERY EXECUTION TIME PREDICTION

Yalçın YENİGÜN

July 2018

LOAD RELATED FEATURE ENGINEERING FOR QUERY EXECUTION TIME PREDICTION

(SORGULARIN ÇALIŞMA SÜRESİNİN TAHMİNİ İÇİN YÜKLE İLİŞKİLİ ÖZNİTELİK MÜHENDİSLİĞİ)

by

Yalçın YENİGÜN, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

in the

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

of

GALATASARAY UNIVERSITY

July 2018

This is to certify that the thesis entitled

LOAD RELATED FEATURE ENGINEERING FOR QUERY EXECUTION TIME PREDICTION

prepared by **Yalçın YENİGÜN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering** at the **Galatasaray University** is approved by the

Examining Committee:

Assist. Prof. Atay ÖZGÖVDE (Supervisor)
Department of Computer Engineering
Galatasaray University

Assoc. Prof. Özlem DURMAZ İNCEL
Department of Computer Engineering
Galatasaray University

Assist. Prof. Berk GÖKBERK
Department of Computer Engineering
MEF University

Date: 05.07.2018

ACKNOWLEDGEMENTS

I would like to thank my supervisors Atay Özgövde and Özlem Durmaz İncel and my company iyzico for their contribution throughout the work on this thesis.

I also owe thanks to my wife Ezgi Hancı Yenigün, M.D. for her unconditional love.

July 2018
Yalçın YENİGÜN

TABLE OF CONTENTS

LIST OF SYMBOLS v
LIST OF FIGURES........ vi
LIST OF TABLES viii
ABSTRACT ix
RÉSUMÉ x
ÖZET........ xi
1. INTRODUCTION........ 1
2. LITERATURE REVIEW 4
2.1 Related Work........ 4
2.2 Previous Feature Engineering Strategies 6
3. MATERIALS AND METHODS........ 10
3.1 Dataset Information 10
3.2 Materials and Experiments........ 12
3.2.1 H2O Framework........ 12
3.2.2 Splunk Database Stream........ 14
3.2.3 MySQL Query Optimizer........ 16
3.2.4 Random Forest Algorithm 20
3.2.5 Linear Regression Algorithm........ 21
3.2.6 Experimental Setup 22
3.3 Evaluation Metrics........ 25
3.4 Feature Engineering for Query Execution Time Prediction........ 26
3.4.1 Query Plan Features 26
3.4.2 Load Features........ 29
4. RESULTS........ 33
4.1 Random Forest Results 33
4.2 Linear Regression Results........ 38

5. CONCLUSION 40
REFERENCES 42

LIST OF SYMBOLS

QET : Query Execution Time
RF : Random Forest
RMSE : Root Mean Square Error
PCA : Principal Component Analysis
CCA : Canonical Correlation Analysis
KCCA : Kernel Canonical Correlation Analysis
MAE : Mean Absolute Error
GLM : Generalized Linear Model
DBPSB : Dpedia Sparql Benchmark

LIST OF FIGURES

Figure 3.1: Maximum Query Execution Times of the System in 24 hours.................. 11
Figure 3.2: Each node builds a subset of random forest in DRF 13
Figure 3.3: H2O Framework Architecture 14
Figure 3.4: Splunk Database Stream Distributed Deployment Architecture 15
Figure 3.5: Example SQL Query for Visual Explain.. 17
Figure 3.6: Visual Explain of An Example Query.. 17
Figure 3.7: JSON Output of an EXPLAIN Query 18
Figure 3.8: Python Code to Remove Duplicated JSON keys 19
Figure 3.9: Python Code to Train and Validate a Random Forest Regression Model... 23
Figure 3.10: Scoring History Deviance Relationship 24
Figure 3.11: The Constant Index Example SQL Query 27
Figure 3.12: The percentages of join types (Const, All, Ref, Index) in the data set...... 27
Figure 3.13: Python Code for Time Series Aggregation with the Given Window Size 29
Figure 3.14: The relation between the total number of bytes transferred and query execution time for 10.000 random instances 30
Figure 3.15: Query for Retrieving Bytes Received and Bytes Sent Information (Global Level)... 31
Figure 3.16: Query for Retrieving Bytes Received and Bytes Sent Information (Session Level)... 31
Figure 3.17: The linear relation between Bytes In and Bytes Out for 10.000 random instances 31
Figure 4.1: RMSE metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads 34
Figure 4.2: R2 metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads 35

Figure 4.3: MAE metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads 35
Figure 4.4: The scaled importance of query plan features and load features together in the highest workload 36
Figure 4.5: Actual vs predicted query execution time in seconds in high workload with query plan features and load features for fast running queries..................................... 37
Figure 4.6: Actual vs predicted query execution time in seconds in high workload with query plan features and load features for slow running queries 37
Figure 4.7: RMSE metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads with Linear Regression 38
Figure 4.8: Standardized Coefficient Magnitudes in Highest Workload for All Features in Generalized Linear Model Algorithm with Linear Regression 39

LIST OF TABLES

Table 3.1: Query Dataset with Different Workloads 10
Table 3.2: Dataset Column Value Distribution in Low Workload 11
Table 3.3: Dataset Column Value Distribution in Medium Workload 11
Table 3.4: Dataset Column Value Distribution in High Workload.............................. 12
Table 3.5: Dataset Column Value Distribution in Highest Workload 12
Table 3.6: Query Plan Features... 26
Table 3.7: Select Types 28
Table 3.8: Load Features 30
Table 3.9: Aggregated Continuous Load Features.. 32

ABSTRACT

Prediction of query execution time is one of the most challenging issues for relational databases and is useful for database administration, resource management, system monitoring and query scheduling. Most of the query optimizers use cost-based models for query execution time prediction but the problem is more complex because the heterogeneity of the database system's hardware platforms and operating systems makes more difficult to measure CPU and I/O costs. The relational database vendors try to implement autonomous databases which automates management and performance thus intelligent query execution time prediction is a key issue. Previous work mostly used synthetical data so that reproducing machine learning experiments are almost impossible for various domains. In this thesis, we use real-world data of a payment service provider with different workloads and we propose new sets of features based on aggregating the database queries and compared them with traditional query plan features. We collected data from a common machine data tool so that reproducing machine learning experiments and building models are easy for various domains.

RÉSUMÉ

La prédiction du temps d'exécution des requêtes est l'un des problèmes les plus complexes pour les bases de données relationnelles et est utile pour l'administration de la base de données, la gestion des ressources, la surveillance du système et la planification des requêtes. La plupart des optimiseurs de requêtes utilisent des modèles basés sur les coûts pour la prédiction du temps d'exécution des requêtes mais le problème est plus complexe car l'hétérogénéité des plateformes matérielles et des systèmes d'exploitation du système de base de données rend plus difficile la mesure des coûts CPU et E/S. Les fournisseurs de bases de données relationnelles essaient d'implémenter des bases de données autonomes qui automatisent la gestion et les performances. La prédiction intelligente de l'heure d'exécution des requêtes est donc un problème majeur. Les travaux antérieurs utilisaient principalement des données synthétiques, de sorte que les expériences de reproduction automatique sont presque impossibles pour divers domaines. Dans cette thèse, nous utilisons des données réelles d'un fournisseur de services de paiement avec différentes charges de travail et nous proposons de nouveaux ensembles de fonctionnalités basées sur l'agrégation des requêtes de base de données et les comparons aux fonctionnalités traditionnelles des plans de requête. Nous avons collecté des données à partir d'un outil commun de données machine afin que les expériences de reproduction automatique et les modèles de construction soient faciles pour différents domaines.

ÖZET

Sorguların çalışma süresini tahmin etmek ilişkisel veri tabanları için en zor konulardan biridir ve bu tahminin doğru gerçekleşmesi, veri tabanı yönetimi, kaynak yönetimi, sistemin performansının izlenmesi ve sorguların zamanlamasının yönetimi gibi birçok konuda faydalıdır. Birçok sorgu iyileştiren yazılım, sorguların çalışma süresini tahmin edebilmek için maliyet tabanlı modeller kullanır fakat ilgili problem daha karmaşıktır zira veri tabanı sistemlerinin donanım ve yazılımlarının heterojen olması işlemci ve G/Ç maliyetlerinin ölçümünü çok zor kılmaktadır. İlişkisel veri tabanı üreticileri, yönetimi ve performansı otomatik hale getiren, kendi kendine çalışan veri tabanı sistemleri geliştirmeye çalışmaktadırlar. Bu noktada veri tabanı sorgularının çalışmadan önce ne kadar süreceğini tahmin etmek kilit bir özelliktir. Geçmiş çalışmalar sorgu süresini tahmin edebilmek için sentetik veri kullanmışlardır. Bu nedenle farklı alanlarda yapay öğrenme deneylerini tekrar etmek neredeyse imkânsız hale gelmektedir. Bu makalede, bir ödeme hizmet sağlayıcısının gerçek dünyadaki farklı yükler altındaki verisi kullanılmış ve veri tabanı sorguları zaman pencereleri içerisinde toplanarak üretilen yeni öznitelik kümesi sunulmuştur. Bu sunulan öznitelik kümesi geleneksel sorgu planı öznitelikleriyle karşılaştırılmış ve sonuçlar paylaşılmıştır. İlgili veri yaygın bir veri toplama aracıyla toplanmış bu sayede yapılan yapay öğrenme deneyleri ve oluşturulan modeller çeşitli alanlarda kolayca tekrar edilebilir hale gelmiştir.

1. INTRODUCTION

In the recent years, the database vendors try to decrease operational cost of database systems with the automation of patch management, high availability approaches and performance tuning. Predicting query execution time is a key issue for automatic intelligent performance tuning. Most of the database systems use cost-based optimizers to predict the query execution time. However, these optimizers are by themselves not sufficient as there are too many operational variables involving the operating system, the hardware characteristics and the database load. The prediction task can become indeterministic and letting too many long running queries can result in system downtime. One of the implicit assumptions behind our work is that the database system's query execution time performance doesn't abruptly change but varies in time and therefore the time series (Hamilton, 1994) analysis is leveraging factor for better prediction.

Recent work on predicting database query execution time has argued that the query optimizer’s cost models are useful to compare alternative queries but not useful to predict database query performance metrics. Ganapathi et al. extracted features before the queries execute and predicts multiple resource usage characteristics for both short and long running queries. They give operator cardinalities and performance metrics to KCCA algorithm and the algorithm interpolates on the relative differences between these cardinalities and builds a model. The training and validation dataset contains queries which have query execution times range from milliseconds to hours and the predicted metrics are elapsed time, records used, disk I/O and message bytes. In that work Ganapathi et al. (2009) designed an architecture which requires vendor site installation for predictions and uses synthetic data which has 2807 instances at maximum.

Moreover, current data management systems process concurrent queries in heterogeneous query workloads and the query performance can be related to the workload size. J. Duggan et al. presents a modeling approach which tries to estimate the effect of concurrency for analytical workloads without using semantic information and adapts the baseline system to dynamically changing workload by using time series analysis. They also introduce a metric that captures the joint effects of disk and memory contention on query performance and predict the latency of each query in the workload and determine the termination time of each query (Duggan et al., 2011).

Previous work mostly uses generated queries from templates but the database management systems can behave differently in production environments. The current systems have database connection pools and these connection pools have limited capacity. The connection pool cannot give connections to applications if there is a latency or too many long-running queries in the system so that identifying long-running queries before execution will help us to better schedule queries. The prediction of query execution time also helps us to know resource requirements of the database management systems before installation.

In high available and scalable systems, latency and failover management is a key issue for scalability because the systems can have latency issues in high workloads. Under high workload, each thread can wait too long and throw timeout exceptions when there are too many long running queries in the system thus there can be a downtime issue because of the constant size of thread pools.

Predicting query execution time can be useful for many system management decisions including:

- System Administration: Knowing query execution time before they are executed can enable cost-based decisions and management systems can decide whether they should execute the query or not
- System Monitoring: Monitoring and alert systems can provide execution time information of queries before they are executed so that systems can avoid too many long-running queries.

- System Sizing: The database infrastructure teams can use the query execution time to give optimum hardware to the systems. This information is also useful for dynamic resource allocation in cloud systems.
- Query Scheduling: Latency aware scheduling systems need the query execution time prediction.

The main contributions of this thesis are listed as follows:

- We propose new sets of load related features to predict query execution time.
- Our machine learning experiments are reproducible because every user of the same database logging platform can collect the similar data in their domain.
- Rather than generating queries from templates, we collect a large data set from production environment of a payment platform and consequently the results of the experiments are reliable for this indeterministic prediction task.

The rest of the thesis is organized as follows: In section 2 we describe related work and previous feature engineering strategies. In section 3 we describe traditional query plan features (Chaudhuri, 1998), we briefly introduce load related features and give information about experiments and materials used in these experiments. In section 4 we present our experiments and results. Finally, we conclude the thesis and outline the future work in section 5.

2. LITERATURE REVIEW

2.1 Related Work

Previous work on query execution time prediction are mostly based on comparative evaluation of various machine learning techniques with synthetic query templates for short and long-running queries. They compared regression techniques, clustering techniques, principal component analysis (PCA), canonical correlation analysis (CCA) and kernel canonical correlation analysis (KCCA) to predict query execution time (Ganapathi et al., 2009) (Akdere et al., 2012). However, these studies do not involve any realistic workloads with real-world production data. Some of the work attempt to address concurrent query performance predictions under different analytical query workloads where TPC-H (Council, T. P. P., 2008) templates are employed for generating the queries (Duggan et al., 2011).

Hasan et al. predicted query execution time of SPARQL queries with a machine learning approach. In that work they extracted algebra features by using the frequencies of algebra operators and extracted graph pattern features by clustering structurally similar query patterns. They use the sum of all SLICE operator cardinalities which is the combination of OFFSET and LIMIT operators appearing in the algebra expression. They also used the depth of the algebra tree and the number of triple patterns as additional features. The DBPSB benchmark is used for evaluation of experiments and this benchmark includes 25 query templates which cover most commonly used SPARQL queries (Hasan & Gandon, 2014).

Moreover, some previous work challenges the assumption of insufficiency of query optimizers and calibrates the constants of optimizer’s cost model for prediction. They show that the optimizer’s cost model can be competitive with machine learning systems

if the parameters of optimizer's cost model is optimized. In general the flow of PostgreSQL's cost optimizer can be summarized as follows (Wu et al., 2013):

- Estimate the cardinality of I/O
- Compute the CPU cost by using the information of the I/O cardinality
- Compute the number of accessed pages with the cardinality estimate information
- Compute the I/O cost with the information of accessed pages
- Compute the sum of CPU cost and I/O cost

Wu W. et al. (2013) also helped the interpretation of database management cost models because they preferred optimizing cost model's parameters instead of black-box machine learning approaches. The other advantages of their framework are as follows:

- No Training Data Needed: The machine learning based query execution time predictions systems need training data but their system can work with only ad-hoc queries.
- Interpretable Approach: The machine learning algorithms are difficult to interpret and most of them are black-box approaches. Their system is related to existing paradigm of query optimization in relational databases.
- Lightweight: The profiling step doesn't depend hardware and the system is portable.

In another work, Wu W. et al. (2014) presents uncertainty aware query execution prediction time methodologies in which they provide prediction uncertainty by using different distributions of likely running times. They try a random distribution for CPU cost of processing one tuple to quantify the uncertainty by assuming that the distribution of CPU cost is normal (i.e. Gaussian) for intuitively the CPU speed is likely to be stable and centered around its mean value. For the selectivity of an operator in the query plan, the uncertainties are different than CPU costs while CPU costs have random values but the selectivity values have fixed numbers because they suppose that any cost model doesn't have a perfect selectivity estimator. In this work they show that the distribution of query execution time t(q) for a query q is asymptotically normal so that this reduces the problem to estimate the two parameters of normal distributions and the mean and variance of t(q) (Wu et al. 2014).

In this thesis we collected a large sample of real-world queries of a payment service provider under different workloads. Results of many machine learning studies in this context are difficult to reproduce (Olorisade et al., 2017). In this sense one of our goals is to increase the probability of being reproduced. To address this, we use one common machine intelligence tool to collect the data and the retrieval process of the dataset is the same for all users of this tool.

2.2 Previous Feature Engineering Strategies

Prior feature engineering studies include following techniques to convert a query to a feature vector (Ganapathi et al., 2009):

- The statistics on the SQL text of each query by counting:
    - Number of nested sub-queries,
    - Total number of selection predicates
    - Number of equality selection predicates
    - Number of non-equality selection predicates
    - Total number of join predicates
    - Number of equijoin predicates
    - Number of aggregation columns
    - Number of sort columns

- Query plan which is produced by the query optimizer (Ganapathi et al., 2009; Akdere et al., 2012; Wu et al., 2013). Query plan feature vector consists of a tree of query operators with estimated cardinalities and contains an instance count and cardinality sum of each possible operator (Ganapathi et al., 2009). Previous work uses PostgreSQL database's query plan features as below (Akdere et al., 2012):
    - seq_page_cost: I/O cost to sequentially access a page which is calculated by query optimizer. Example query:
        - **SELECT * FROM** R;
    - random_page_cost: I/O cost to randomly access a page. Example query:
        - **SELECT * FROM** R **WHERE** R.B < b; (b is unclustered index)

- cpu_tuple_cost: CPU cost to process a tuple which is unordered sets of known values with names. Example query:
  - **SELECT * FROM** R;
- cpu_index_tuple_cost: CPU cost to process a tuple via index. Example query:
  - **SELECT * FROM** R **WHERE** R.A < a; (a is clustered index)
- cpu_aggregator_cost: CPU cost of aggregation or hash which is calculated by query optimizer. Example query:
  - **SELECT COUNT**(*) **FROM** R;

- Akdere et al. (2012) use query plan level features which is calculated by query optimizer as follows:
  - **p_tot_cost:** Estimated total plan cost
  - **p_st_cost:** Estimated plan start cost
  - **p_rows:** Estimated number of output tuples
  - **ap_width:** Estimated average size of an output tuple
  - **op_count:** Number of query operators in the plan
  - **row_count:** Estimated total number of tuples input and output to/from each operator
  - **byte_count:** Estimated total size of all tuples input and output
  - **<opertor_name>_count:** The number of operators in the query
  - **<operator_name>_rows:** The total number of tuples output from <operator_name> operators

- Akdere et al. (2012) also used operator level features which is collected by using “MATERIALISE” operator. The operator level features use two separate prediction models as start-time model and run-time model. The start-time model tries to estimate the execution time of an operator until it produces its first tuple. The run-time model tries to estimate total execution time of query operators:
  - **np:** Estimated I/O (in number of pages)
  - **nt:** Estimated number of output tuples
  - **nt1:** Estimated number of input tuples (from left child operator)

- **nt2:** Estimated number of input tuples (from left right operator)
- **sel:** Estimated operator selectivity
- **st1:** Start-time of left child operator
- **rt1:** Run-time of left child operator
- **st2:** Start-time of right child operator
- **rt2:** Run-time of right child operator

- The structural similarity between graph queries by using clustering techniques. They construct multiple graphs from multiple query patterns and compute the edit distance between these graphs to compute the structural similarity between query patterns (Hasan & Gandon, 2014).

- Wu W. et al. (2013) used queuing theory and they designed a queuing model which uses prediction pipelines of multiple queries as customers. In this model the query execution time of a pipeline is its residence time in queueing network. If k ∈ {CPU, disk}, the features of this queuing model are as follows:
  - **C(k):** Number of servers in (service) center k
  - **T(k):** Mean service time per visit to center k
  - **Y(k):** Correction factor of center k
  - **p(k):** Utility of center k
  - **V(k,m):** Mean number of visits by customer m to center k
  - **Q(k,m):** Mean queue length by customer m at center k
  - **R(k,m):** Mean residence time per visit by customer m to center k

- We W. et al (2013) also uses scan operator features like sequential scan, index scan and bitmap index scan which are implemented by PostgreSQL. They use the following features to represent a scan instance *s(i)* in a mix {s1 , ..., sn}, where *tbl(i)* is the table accessed by *s(i),* and *N(s(i))* is the set of neighbor scans of *s(i)* in the mix:
    - Number of sequential I/O’s of s(i)
    - Number of random I/O’s of s(i)
    - Number of scans in N(s(i)) that are over tbl(i)
    - Number of sequential I/O’s from scans in N(s(i)) that are over tbl(i)
    - Number of random I/O’s from scans in N(s(i)) that are over tbl(i)
    - Number of scans in N(s(i)) that are not over tbl(i)
    - Number of sequential I/O’s from scans in N(s(i)) that not are over tbl(i)
    - Number of random I/O’s from scans in N(s(i)) that are not over tbl(i)

3. MATERIALS AND METHODS

3.1 Dataset Information

The data set is collected for 40 minutes from a payment service provider with a database plugin of streaming machine data platform in four different workloads. The highest workload data is collected during black Friday between 21.55 and 22.05, the lowest workload data is collected in the early morning between 03.00 and 03.10 (See Table 3.1). The data set contains all queries of the company from too many different schemas and applications with the limit of 10 minutes overall execution time. In Table 3.1, minimum, maximum and mean of query execution times in related workload type is shown.

Table 3.1: Query Dataset with Different Workloads

Workload Type	Number of Instances	Minimum QET	Mean QET	Maximum QET
Low	199789	0.0001 secs	0.01 secs	33.5 secs
Medium	425847	0.0001 secs	0.009 secs	41.5 secs
High	425989	0.001 secs	0.05 secs	152 secs
Highest	944254	0.0001 secs	0.006 secs	8.5 secs

The dataset has 4 distinct samples and each sample has 10 minutes window size with different workload sizes. When we look at the behavior of the overall system in 24 hours, the maximum query execution times change between 10 seconds and 150 seconds (Figure 3.1). The mean of the query execution times changes between 0.006

seconds and 0.05 seconds. The highest workload data sample size is 780 MB while the lowest workload data sample size is 140 MB.

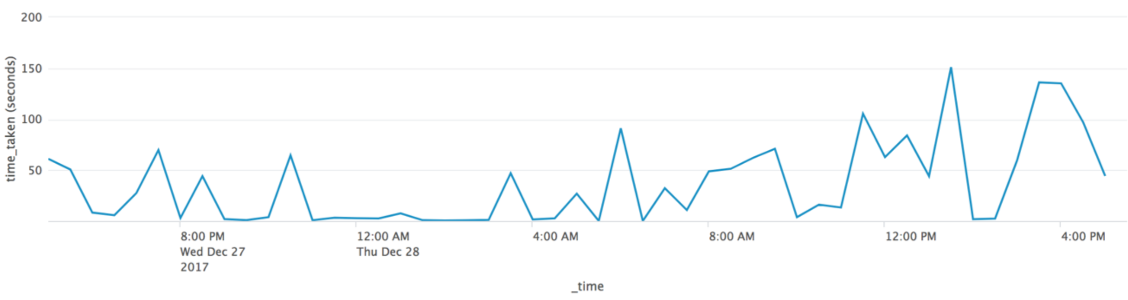

Figure 3.1: Maximum Query Execution Times of the System in 24 hours

The minimum, maximum and mean of other numerical column values are shown in tables for all workloads (Table 3.2, 3.3, 3.4 and 3.5).

Table 3.2: Dataset Column Value Distribution in Low Workload

Column Name	Workload Type	Minimum	Mean	Maximum
Reply Time	Low	0	0.01	33.5
Request Time	Low	0	0.00002	0.2
Bytes	Low	24	3090	4399465
Bytes In	Low	13	1096	20835
Bytes Out	Low	0	1993	4378630
Result Column Count	Low	1	6.8	182
Result Row Count	Low	0	1	18

Table 3.3: Dataset Column Value Distribution in Medium Workload

Column Name	Workload Type	Minimum	Mean	Maximum
Reply Time	Medium	0	0.008	41.5
Request Time	Medium	0	0.00001	0.3
Bytes	Medium	24	3319	2104348
Bytes In	Medium	13	1153	20835
Bytes Out	Medium	0	2165	2103717
Result Column Count	Medium	1	6	114
Result Row Count	Medium	0	1	30

Table 3.4: Dataset Column Value Distribution in High Workload

Column Name	Workload Type	Minimum	Mean	Maximum
Reply Time	High	0	0.04	152
Request Time	High	0	0.00001	0.3
Bytes	High	24	3090	8886522
Bytes In	High	13	804	60199
Bytes Out	High	11	2364	8885920
Result Column Count	High	1	7	114
Result Row Count	High	0	1	23

Table 3.5: Dataset Column Value Distribution in Highest Workload

Column Name	Workload Type	Minimum	Mean	Maximum
Reply Time	Highest	0	0.06	8
Request Time	Highest	0	0.00005	21.5
Bytes	Highest	24	1713	8035174
Bytes In	Highest	13	595	20835
Bytes Out	Highest	0	1118	8032750
Result Column Count	Highest	1	7	122
Result Row Count	Highest	0	1	100

3.2 Materials and Experiments

3.2.1 H2O Framework

H2O framework is open-source, in-memory and distributed machine learning framework for big data analytics. Inside the framework, a distributed key value store is used for data manipulation and model building so that each node has an access to data frames and models (Figure 3.3). The framework has three parts:

- **H2O Cluster:** Multi-node cluster with shared memory model
- **Distributed Key-Value Store:** Data frames, models and any other H2O objects can be accessed via distributed key value store

- **H2O Frame:** Distributed data frames which their columns are distributed (across the nodes) arrays

In the experiments, Distributed Random Forest (DRF) (Segal, 2004) algorithm is used and this algorithm can handle categorical variables and missing values automatically. In the DRF algorithm of H2O framework, missing values are not interpreted as missing at random. They are interpreted as containing information (i.e., missing for a reason). Split decisions for every node are found by minimizing the loss function and treating missing values as a separate category that can go either left or right during tree building. In DRF each node builds a subset of forest (Figure 3.2) (Liaw & Wiener, 2002).

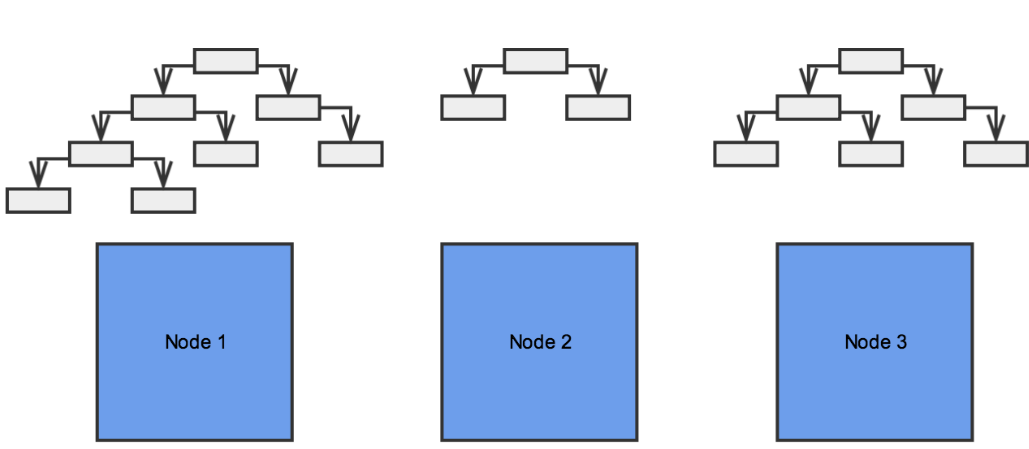

Figure 3.2: Each node builds a subset of random forest in DRF

The H2O Framework has different components working together to build machine learning models (Figure 3.3). The data scientists or software developers can build models by using different programming languages like Python, R, JavaScript ..etc. and the framework is also integrated with business intelligence tools. The framework also provides plugins for different data sources from relational databases to big data tools and it helps us to run experiments with millions of queries with its scalability and data compression capabilities.

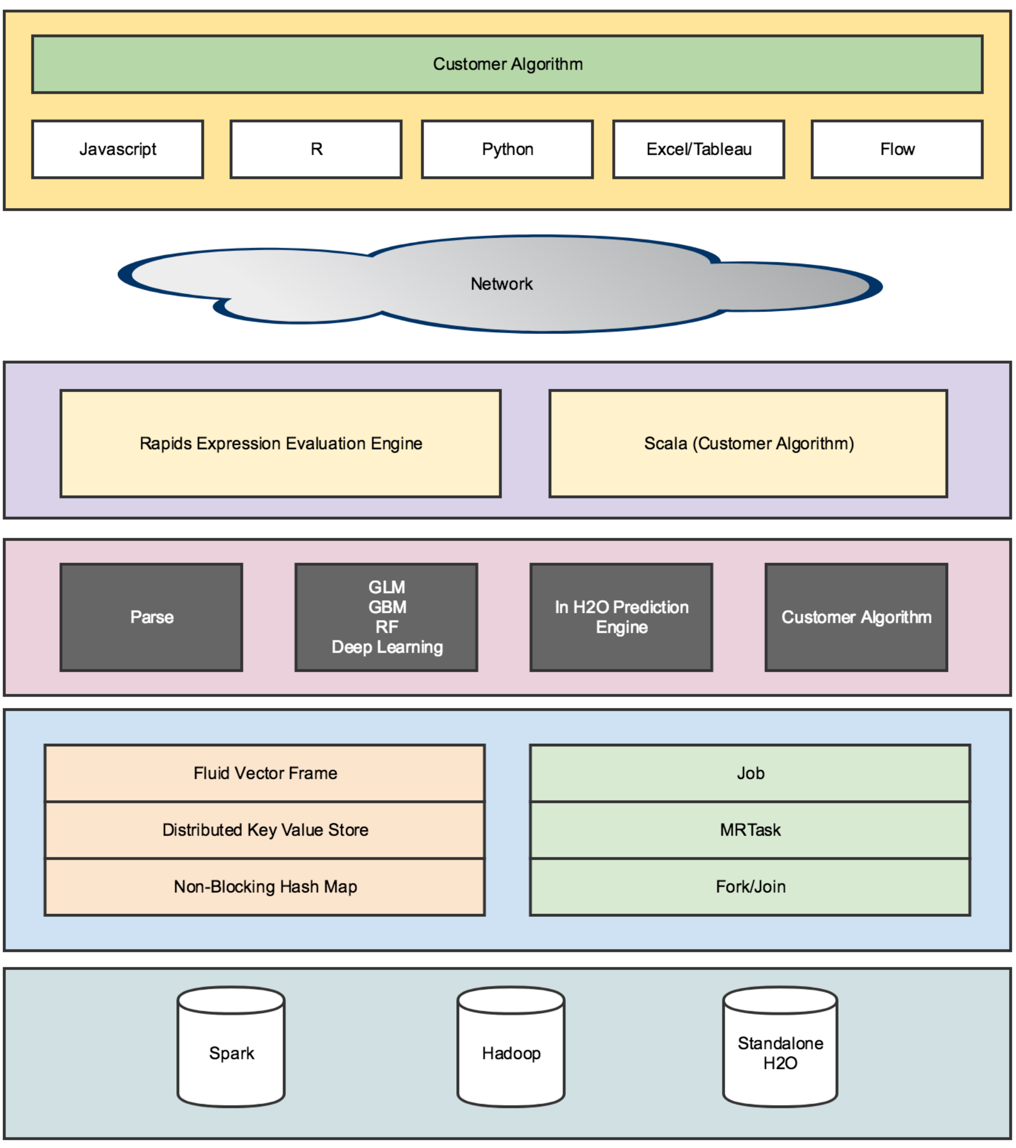

Figure 3.3: H2O Framework Architecture

3.2.2 Splunk Database Stream

Splunk is a machine data collection tool for distributed applications and the query dataset is collected by using Splunk database stream. Streams support passive capture of network data and database protocol is one of the supported protocols of streams. To collect data from a database, the forwarders forward the query data and the execution times to indexer and the indexer indexes the queries for searching.

Splunk database stream supports MySQL database and can be installed on a single instance server as both search head and indexer which is ideal for small local and testing environment because it supports one or two concurrent searches. For large datasets and enterprise applications, distributed installation is needed (Figure 3.4).

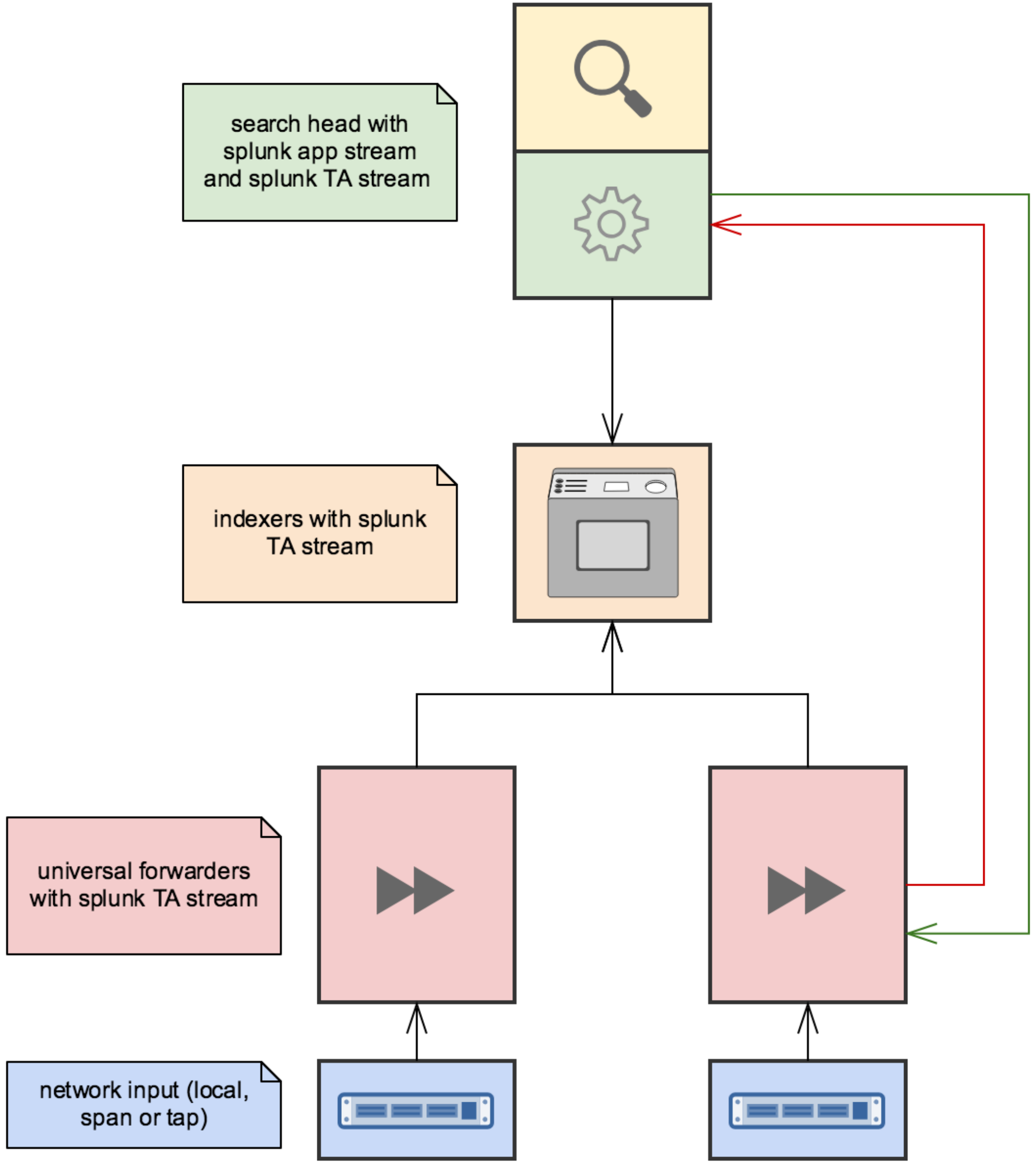

Figure 3.4: Splunk Database Stream Distributed Deployment Architecture

To collect queries and their execution times in a scalable and distributed architecture, the database queries are collected as network inputs by universal forwarders of a database stream. The universal forwarders of the plugin collect the query, the query parameters and their execution time and forward this information to indexers. To find all the queries, the head pointer points the indexer so that the search results appear on the user interface of plugin (Figure 3.4).

3.2.3 MySQL Query Optimizer

MySQL query optimizer tries to optimize the queries in database level and hardware level. In the database level the optimizer looks at the database's basic design:

- Do the columns have the right data types?
- Is the right storage engine (MyIsam or InnoDB) used? For example, if the system is transactional, it needs a transactional storage engine InnoDB.
- Are the right indexes in place to make query executions faster?
- Is the application uses the right locking strategy? For example, InnoDB storage engine guarantees data consistency.
- Is the memory used efficiently for caching the queries?

In the hardware level the optimizer looks at the hardware limits:

- CPU cycles: Having large tables need more memory.
- Memory bandwidth: If CPU needs more memory than the size of CPU cache, the main memory bandwidth can be a bottleneck.
- Disk seeks: The disk needs time to find a piece data.
- Disk reading and writing: Reading data from the disk and writing the data to the disk may be expensive.

In MySQL database, the query plan information can be retrieved by using EXPLAIN query. In Figure 3.5, an example SQL query is given with selection, concatenation, inner joins and where operation with some limit. This query has a full table scan cost and multiple nested loop costs which uses unique key and non-unique key lookups. MySQL query plan calculates the cost of the query by using these parameters (Figure 3.5).

```
SELECT CONCAT(customer.last_name, ', ',
customer.first_name) AS customer, address.phone, film.title
FROM rental
INNER JOIN customer ON rental.customer_id =
customer.customer_id
INNER JOIN address ON customer.address_id =
address.address_id
INNER JOIN inventory ON rental.inventory_id =
inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY <
CURRENT_DATE()
LIMIT 5;
```

Figure 3.5: Example SQL Query for Visual Explain

The visual information about the explain is as follows (Figure 3.6):

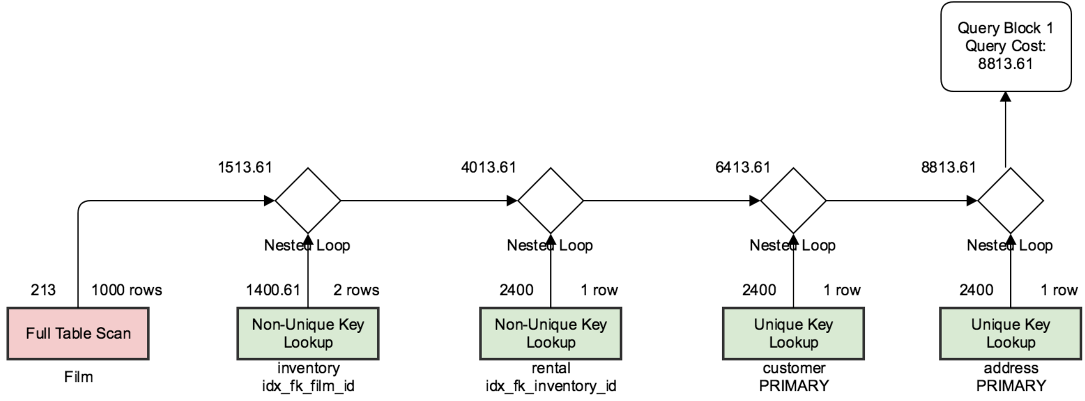

Figure 3.6: Visual Explain of An Example Query

The MySQL optimizer uses disk seek count to estimate the query performance in hardware level:

$$\frac{\log(row_count)}{\frac{\log(index_block_length)}{3} \times \frac{2}{index_length + data_pointer_length}} + 1 \quad (1)$$

The query plan features are collected by using MySQL's "EXPLAIN JSON" query. In MySQL's EXPLAIN query, the output JSON format (Figure 3.7) is not valid because there are many keys which have the same names (i.e. table, table_name ..etc.):

```
EXPLAIN: {
  "query_block": {
    "select_id": 1,
    "nested_loop": [
      {  "table": {
          "table_name": "departments",
          <skipped>
      },
      {  "table": {
          "table_name": "<subquery2>",
          "access_type": "eq_ref",
          "key": "<auto_key>",
          "key_length": "4",
          "ref": [
            "employees.departments.dept_no"
          ],
          "rows_examined_per_scan": 1,
          "materialized_from_subquery": {
            "using_temporary_table": true,
            "query_block": {
              "table": {
                "table_name": "dept_manager",
                "access_type": "ALL",
                "possible_keys": [
                  "dept_no"
                ], "used_columns": [
                     "dept_no",
                     "to_date"
                ]}}}}}]}
}
```

Figure 3.7: JSON Output of an EXPLAIN Query

To fix the invalid JSON format problem, we have to differentiate key names by adding postfixes to the names. We use Python programming language in all experiments and fix invalid JSON problem with the following code (Figure 3.8) :

```
from collections import OrderedDict
from json import JSONDecoder

def make_unique(key, dct):
    counter = 0
    unique_key = key

    while unique_key in dct:
        counter += 1
        unique_key = '{}_{}'.format(key, counter)
    return unique_key


def parse_object_pairs(pairs):
    dct = OrderedDict()
    for key, value in pairs:
        if key in dct:
            key = make_unique(key, dct)
        dct[key] = value

    return dct
```

Figure 3.8: Python Code to Remove Duplicated JSON keys

We give the EXPLAIN output to *parse_object_pairs* function to make the JSON keys unique so that multiple *table* keys are renamed as *table_1, table_2..., table_n.*

3.2.4 Random Forest Algorithm

Growing ensemble of trees and letting them vote for the most popular class in classification resulted improvements in classification accuracy. Random vectors are created for growing the ensembles of trees. Random forest algorithm is also used for regression and we make our experiments with this algorithm. Random forests for regression are created by growing trees depending on a random vector such that the predictor trees take on numerical values rather than categorical values. Each leaf of random trees contains a distribution for the continuous output variable. The random forest predictor is created by taking the average of k of the trees for final prediction. Some of the input features may be categorical and since the random forest algorithm wants to define additive combinations of variables, the algorithm needs to define how categorical variables will be treated so they can be combined with numerical variables (Breiman, 2001).

Random forest algorithm is that each time a categorical feature is selected to split on at a node, to select a random subset of the categories of the variable, and define a substitute variable that is one when the categorical value of the variable is in the subset and zero outside (Breiman, 2001). In H2O framework, random forest algorithm handles categorical variables with the following encoding types:

- **Auto:** Allow the algorithm to decide and it is the default encoding scheme to handle categorical features. In H2O random forest the algorithm will automatically perform Enum encoding.
- **Enum:** The algorithm adds one column per categorical feature.
- **One Hot Explicit:** The algorithm adds N+1 new columns for categorical features with N levels.
- **Binary:** The algorithm adds no more than 32 columns per categorical feature.
- **Eigen:** The algorithm adds k columns per categorical feature, keeping projections of one-hot-encoded matrix onto k-dimension eigen space only.
- **Label Encoder:** The algorithm converts every enumeration into the integer of its index (for example, level 0 -> 0, level 1 -> 1, etc.).

In H2O framework, random forest algorithm splits the trees based on reduction in Squared Error for regression. For categorical features, the framework also uses histograms for splitting and can handle splitting on categorical variables with the chosen encoding type. In the experiments, we used "enum" encoding which adds one column per categorical feature to split on categorical feature and assigns 1 if the instance is in this category, 0 if the instance is not in this category.

3.2.5 Linear Regression Algorithm

Linear regression algorithm tries to model the relationship between two or more explanatory variables by setting a linear equation to observed data. One variable is considered to be explanatory, and the other variable is considered to be dependent. It is the simplest sample of a linear algorithm but has many uses and several advantages over other algorithms. Especially, it is faster and requires more stable computations. H2O framework handles categorical variables automatically by expanding them into one-hot encoded binary vectors. The framework creates variables which is a variable created to assign numerical value to levels of categorical variables and each variable represents one category of the explanatory variable and is coded with 1 if the case falls in that category and with 0 if not. Consequently the algorithm adds one binary column for each categorical feature and encodes it with zero or one (Seber & Lee, 2012).

Linear regression model parameters in H2O framework are as follows (Ambati et al., 2014):

- **training_frame:** Specify the dataset used to build the model
- **seed:** Specify the random number generator seed for algorithm components dependent on randomization. The seed is consistent for each H2O instance so that you can create models with the same starting conditions in alternative configurations.
- **y:** Specify the column to use as the dependent variable. For a regression model, this column must be numeric. For a classification model, this column must be categorical (Enum or String).

- **x:** Specify a vector containing the names or indices of the predictor variables to use when building the model. If x is missing, then all columns except y are used.
- **alpha:** Specify the regularization distribution between L1 and L2.
- **lambda:** Specify the regularization strength.
- **early_stopping:** Specify whether to stop early when there is no more relative improvement on the training or validation set.
- **standardize:** Specify whether to standardize the numeric columns to have a mean of zero and unit variance. If the standardization is not used, the results can include components that are dominated by variables that appear to have larger variances relative to other attributes as a matter of scale, rather than true contribution. This option is enabled by default and we used this parameter in the experiments.
- **max_iterations:** Specify the number of training iterations. It was 100 in the experiments.
- **objective_epsilon:** Specify a threshold for convergence. The model is converged if the objective value is less than this threshold.
- **beta_epsilon:** Specify the beta epsilon value. If the L1 normalization of the current beta change is below this threshold, consider using convergence.

3.2.6 Experimental Setup

All the experiments used MySQL version 5.5 servers. The experiment infrastructure is a DELL R730 eight processor machines with Emc5100 disks and 8 GB RAM. The data is collected by using MySQL database plugin of Splunk Enterprise version 6.6.0. Machine learning training and tests are made by using H2O.ai implementation of Random Forest (Liaw & Wiener, 2002) for regression with 100 trees. The Python programming language is used for all experiments. The queries are split as 80% for training and 20% for testing and the same queries are not used for both training and testing. Window size is 30 seconds for load feature aggregation. We predict query execution time in seconds and measure error rates. The Random Forest parameter values are shown below:

- Number of Trees: 100
- Number of Internal Trees: 100
- Minimum Depth: 20
- Maximum Depth: 20
- Mean Depth: 20
- Minimum Leaves: 2492
- Maximum Leaves: 10353
- Categorical Encoding: Enum

Random forest algorithm (Breiman, 2001) adds additional layer of randomness to bagging which successful trees don't depend on earlier trees. In standard decision trees, each node of the tree is split by the best split in all variables. In a random forest, each node of the tree is split by the best in a random subset of predictors at that node. This randomness makes random forest robust to overfitting thus random forest algorithm is chosen in all experiments. The example Python code to split training data frame and test data frame and train a DRF model is shown below (Figure 3.9) :

```
from h2o import h2o
h2o.init()
h2o.remove_all()

from h2o.estimators.random_forest import H2ORandomForestEstimator

train, valid = no_peak_df_plan_h2o.split_frame(ratios = [.8], seed = 1234)

feature_columns.extend(continuous_feature_columns)

random_forest= h2o.H2ORandomForestEstimator(
    model_id="rf_no_peak_all",
    ntrees=100
)
random_forest.train(feature_columns, 'time_taken', training_frame=train,
validation_frame=valid)
```

Figure 3.9: Python Code to Train and Validate a Random Forest Regression Model

The validation (Picard & Cook, 1984) data set deviance decreases with the increase of the number of trees in the Random Forest (Figure 3.10). The deviance is the goodness of fit in statistics so that if it decreases, the model predicts better. In this model, the deviance becomes constant at 80 trees and increasing the number of trees more than 80 doesn't help to increase accuracy so that we can decrease computation resource of predictions if we use only 80 trees. Decreasing number of trees in a Random Forest algorithm helps us to use less hardware resources for computations and to decrease training and prediction times so that we measure the deviance to fix the number of trees in each training.

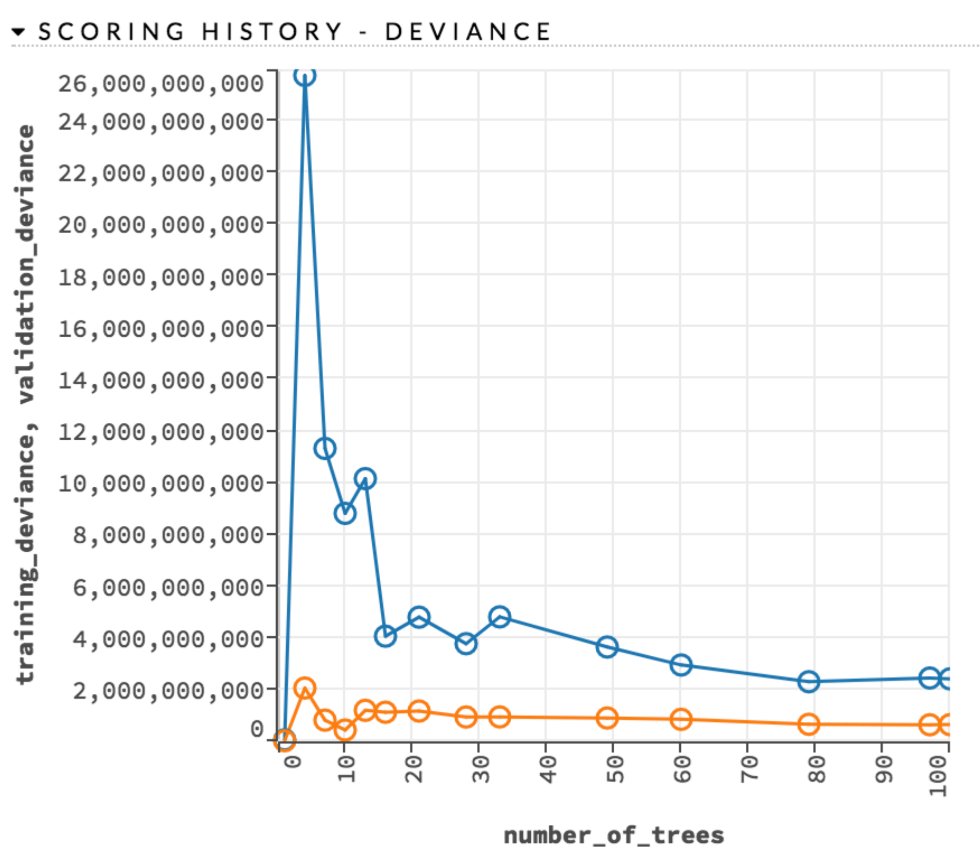

Figure 3.10: Scoring History – Deviance Relationship

In the experiments we also try to predict query execution time with Generalized Linear Model (GLM) of H2O framework with Linear Regression. The linear regression model

predictions take the form of a full predictive distribution. Linear Regression corresponds to the Gaussian family model (Frisch et al., 1996; Rasmussen, 2004; Bunea et al., 2007).

3.3 Evaluation Metrics

Root Mean Squared Error (RMSE) (1) (Chai & Draxler, 2014) is used as the error metric of experiments. This metric is useful to minimize the absolute difference in actual and predicted query execution times.

$$\text{RMSE}=\sqrt{\frac{1}{n}\sum_{j=1}^{n}(\text{actual-estimate})^2} \quad (2)$$

R2 (2), the coefficient of determination (Nakagawa & Schielzeth, 2013), also called the multiple correlation is used as the second metric. y and y' represent the actual and the predicted values in n queries. As R2 is scaled between 0 and 1, it is easy to interpret the results and to compare the results of different workloads.

$$R^2(y,y')=1-\frac{\sum_{i=1}^{n}(y_i-y'_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2} \quad (3)$$

Mean Absolute Error (MAE) is the average magnitude of the errors in a set of predictions without considering their direction (Willmott & Matsuura, 2005). If all individual differences have the same weight, the mean absolute error is the average over the test dataset of the absolute differences between prediction and actual values (3).

$$\text{MAE}=\frac{1}{n}\sum_{j=1}^{n}\left|y_j-y'_j\right| \quad (4)$$

3.4 Feature Engineering for Query Execution Time Prediction

3.4.1 Query Plan Features

In this section we consider how will the relational database management system executes a query. Our first approach for feature engineering relies on query execution plans since many database management systems provide interfaces that returns query plan information. The query plan features include information about partitions, how tables are joined, the indexes that are chosen and the join types (See Table 3.6). The query plan structure is collected with “EXPLAIN” interface of MySQL database.

Table 3.6: Query Plan Features

Feature Name	Description
Access Type	The join type (equijoin, non-equijoin etc.)
Select Id	The sequential number of select within the query
Select Type	Types of select
Table Name	The name of the table to which the row of output refers
Key	Index that is actually decided to use
Key Length	The length of the index that is actually decided to use
Used Key Parts	The part of a multiple column key that is actually used
Ref	Columns that are compared to the index
Rows	The number of rows that database believes it must examine to execute the query
Using Index	The column information is retrieved using only indexes

The access type can have the values of “all”, “index”, “const” and “ref”:

- **ALL:** For each combination of rows from the previous tables, a full table scan is done. By adding indexes that enable row retrieval from the table based on constant values or column values from earlier tables, we can avoid ALL.

- **INDEX:** This join type is the same as ALL, except that the index tree is scanned. This may occur in two ways:

  - If the index is a covering index only the index tree is scanned. In this case, the query plan returns “using index”. An index-only scan usually is faster than ALL because the size of the index usually is smaller than the table data.
  - A full table scan is performed using reads from the index to look up data rows in index order. Uses index does not appear in the extra column.

- **CONST:** The table has one matching row at most and this row is read at the start of the query. These tables are very fast because they are read-only at once. This index type is used when you compare all parts of a primary key or unique key to constant values. In the following queries, my_table can be used as a const table (Figure 3.11).

```
SELECT * FROM my_table WHERE primary_key = 1;

SELECT * FROM my_table WHERE primary_key_part1 = 1 AND
primary_key_part2 = 2;
```

Figure 3.11: The Constant Index Example SQL Query

- **REF:** This column describes which columns or constants are compared to the index named in the key column to select rows from the table.

In the dataset the most used join types are Const and All. 45% of queries uses Const, 31.1% of queries uses All, 21.2% of queries uses Ref and 2.68% of queries uses Index (Figure 3.12).

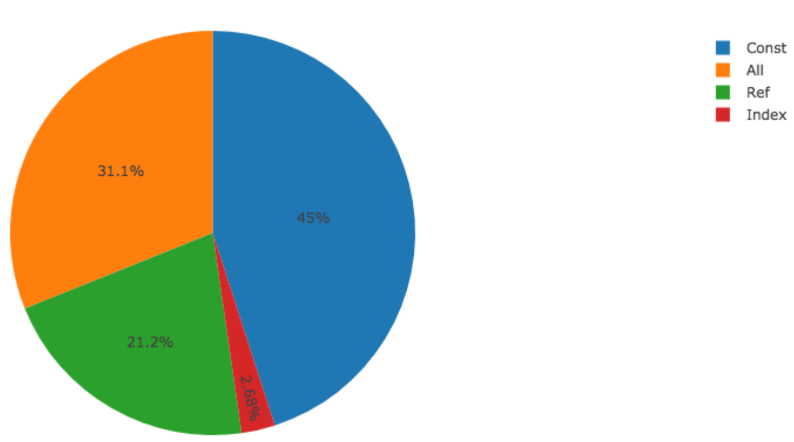

Figure 3.12: The percentages of join types (Const, All, Ref, Index) in the data set

The select types of EXPLAIN output are shown in Table 3.7:

Table 3.7: Select Types

Select Type	Description
SIMPLE	Simple SELECT (not using sub queries or UNION)
PRIMARY	Outermost SELECT
UNION	Second or later SELECT statement in a UNION
DEPENDENT UNION	Second or later SELECT statement in a UNION, dependent on outer query
UNION RESUT	Result of a UNION query
SUBQUERY	First select in a subquery
DEPENDENT SUBQUERY	First SELECT in subquery, dependent on outer query
DERIVED	Derived table SELECT (subquery in FROM clause)
MATERIALIZED	Materialized subquery
UNCACHEABLE SUBQUERY	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
UNCACHEABLE UNION	The second or later select in a UNION that belongs to an uncacheable subquery

3.4.2 Load Features

The second feature engineering methodology is based on analyzing the periodic behavior of a database management system by aggregating the queries with time series analysis. The data is collected by using a common database plugin of a machine data collection tool. We aggregated time features like minimum, maximum, mean, standard deviation and variance with constant window size and analyzed the effects of workload to predict query execution time (See Table 3.8). For example the aggregation of “bytes” column in a rolling window is implemented with Pandas framework as follows (Figure 3.13) :

```
import pandas as pd

# read CSV file from a data path
df = pd.read_csv(data_path, index_col=None, header=0)

# aggregate features in a rolling window
df['bytes_mean']= df['bytes'].rolling(window_size).mean()
df['bytes_std']= df['bytes'].rolling(window_size).std()
df['bytes_var']= df['bytes'].rolling(window_size).var()
df['bytes_min']= df['bytes'].rolling(window_size).min()
df['bytes_max']= df['bytes'].rolling(window_size).max()
```

Figure 3.13: Python Code for Time Series Aggregation with the Given Window Size

Table 3.8: Load Features

Feature Name	Description
Reply Time	Number of microseconds that it took the server to start replying to a request
Request Time	Number of microseconds that it took the client to send a request
Response Time	Number of microseconds that it took the server to send a response
Bytes	The total number of bytes transferred
Bytes In	The number of bytes sent from client to server
Bytes Out	The number of bytes sent from server to client

We try to investigate the relationship between the total number of bytes transferred and the query execution time to better understand the effects of network workload but we couldn't find a linear relation (Figure 3.14).

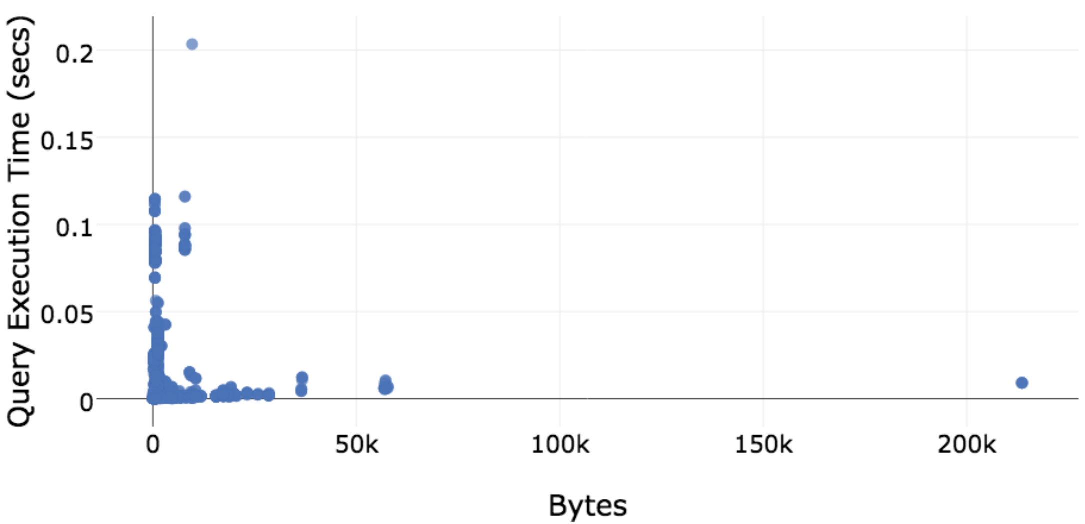

Figure 3.14: The relation between the total number of bytes transferred and query execution time for 10.000 random instances

The number of bytes sent from client to server (bytes in) and the number of bytes sent from server to client (bytes out) depends linearly (Figure 3.17). This information about the number of bytes can be obtained with the query below in MySQL:

```
SELECT * FROM information_schema.global_status
WHERE variable_name IN
('Bytes_received','Bytes_sent');
```

Figure 3.15: Query for Retrieving Bytes Received and Bytes Sent Information (Global Level)

```
SELECT * FROM information_schema.session_status
WHERE variable_name IN
('Bytes_received','Bytes_sent');
```

Figure 3.16: Query for Retrieving Bytes Received and Bytes Sent Information (Session Level)

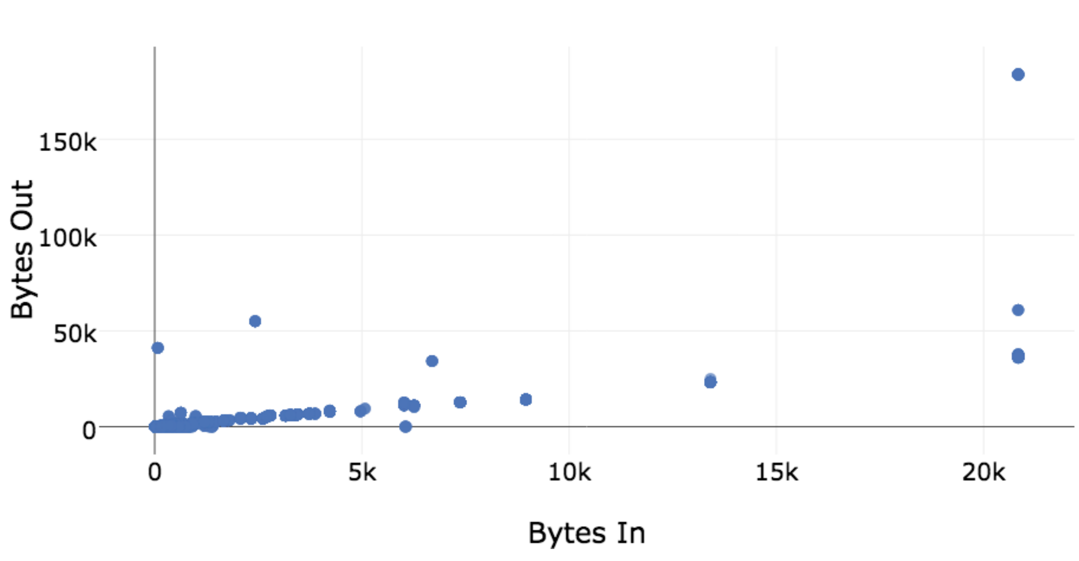

Figure 3.17: The linear relation between Bytes In and Bytes Out for 10.000 random instances

After the aggregation of load features, the continuous aggregated features are shown in Table 3.9 with window size of 30 seconds.

Table 3.9: Aggregated Continuous Load Features

Feature Name	Description
Reply Time Mean	The mean of reply time in the current window
Reply Time Std	The standard deviation of reply time in the current window
Reply Time Var	The variance of reply time in the current window
Reply Time Min	The minimum value of reply time in the current window
Reply Time Max	The maximum value of reply time in the current window
Request Time Mean	The mean of request time in the current window
Request Time Std	The standard deviation of request time in the current window
Request Time Var	The variance of request time in the current window
Request Time Min	The minimum value of request time in the current window
Request Time Max	The maximum value of request time in the current window
Response Time Mean	The mean of response time in the current window
Response Time Std	The standard deviation of response time in the current window
Response Time Var	The variance of response time in the current window
Response Time Min	The minimum value of response time in the current window
Response Time Max	The maximum value of response time in the current window
Bytes Mean	The mean of total number of bytes transferred in the current window
Bytes Std	The standard deviation of total number of bytes transferred in the current window
Bytes Var	The variance of total number of bytes transferred in the current window
Bytes Min	The minimum value of total number of bytes transferred in the current window
Bytes Max	The maximum value of total number of bytes transferred in the current window

4. RESULTS

4.1 Random Forest Results

First, we show the results of predictions with query plan features and load features by using random forest algorithm. We build a random forest regression model to predict query execution time in seconds in different workloads and compare RMSE metric of query plan features, load features and all features together with different workloads (see Figure 4.1).

In Figure 4.1, the x-axis represents different workloads and the y-axis shows the RMSE of the predictions. The best results are obtained with the model of query plan and load features together in all workloads. Surprisingly, for the highest workload (1573 queries per second) during black Friday, query plan features are more effective than load features. We need to interpret random forest algorithm to analyze this result so that we calculate the relative influence of each feature whether this feature was selected during splitting in the tree building process and how much the squared error (over all trees) improved as a result. The scaled importance of features is calculated by h2o.ai framework (Ambati et al., 2014).

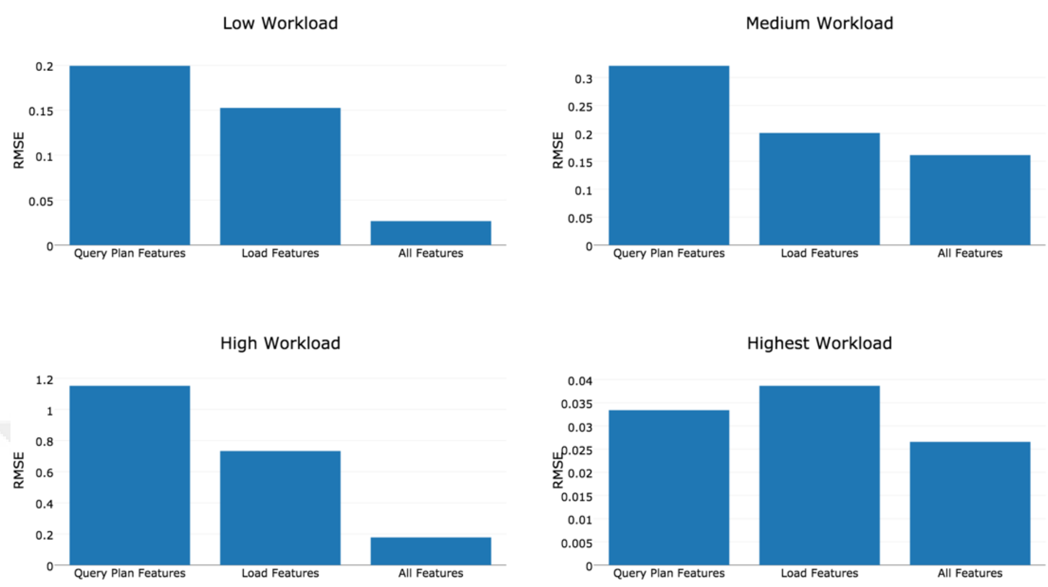

Figure 4.1: RMSE metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads

Figure 4.2 shows the R2 values on the validation dataset for different workloads. The low workload gives us the highest R2 value 0.9821 while the highest workload gives the lowest R2 value 0.5268 with query plan and load features together.

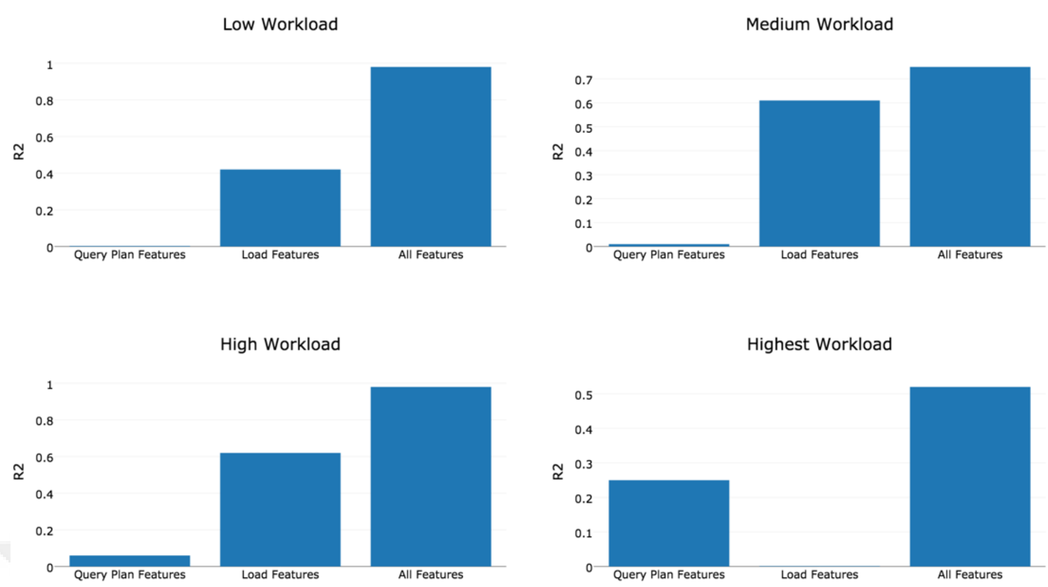

Figure 4.2: R2 metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads

Figure 4.3 shows the mean absolute error (MAE) values on the validation set of different workloads. The low workload gives us the lowest MAE value 1266 and while the high workload gives us the highest MAE value 4878.

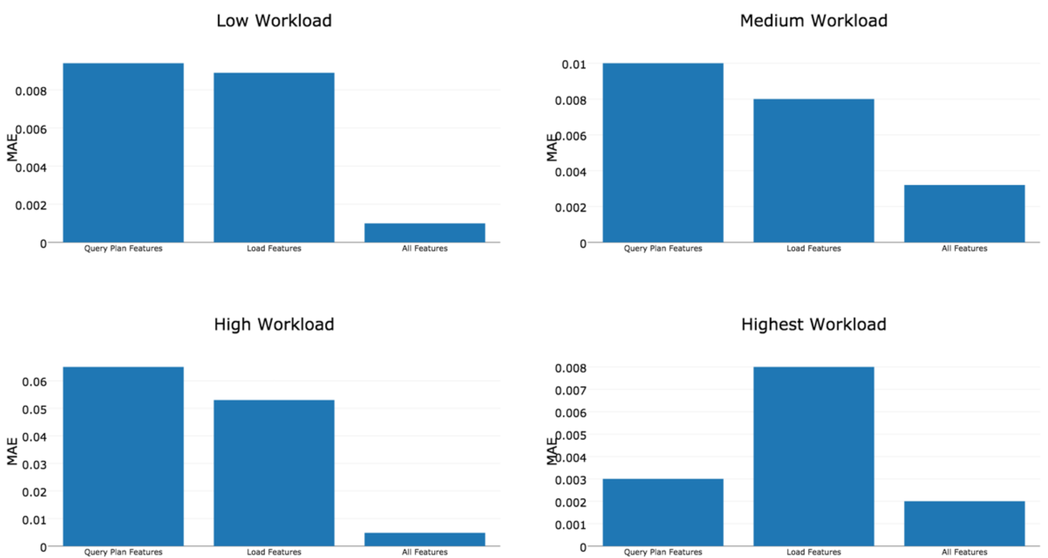

Figure 4.3: MAE metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads

Comparing the scaled importance (Strobl et al., 2007; Menze et al., 2009) of each feature, we observe that the most important features correlate with a particular database table during black Friday. The effect of query plan features in the highest workload is due to the usage of the same table (table_1) under the load (See Figure 4.4). When we analyze only the load features, the mean and variance of total number of bytes transferred in the current window are important features thus we validate the relation between total bytes transferred and query execution time (See Figure 4.4).

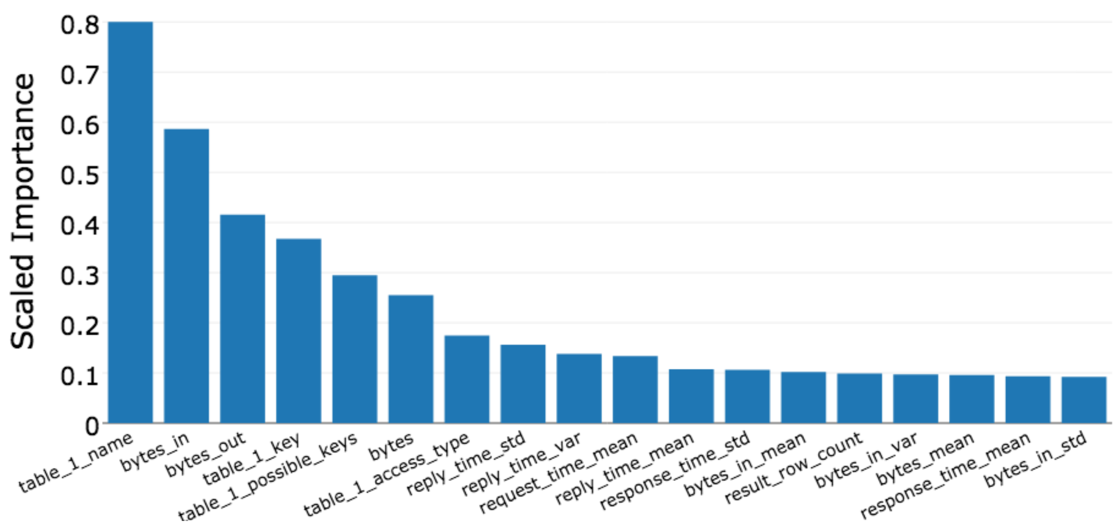

Figure 4.4: The scaled importance of query plan features and load features together in the highest workload

To provide more information about the results, we show the predicted and actual query execution times for a random subset of test queries having size of 15.000 (See Figure 4.5 and Figure 4.6).

We categorize queries by query execution time and create slow and fast running query pools and our predictions are linear for both slow and fast queries while using query plan and load features together. When we look at the results, the predicted and actual execution times are linear thus the query plan features and load features are useful when

they are used together in a machine learning algorithm for both fast (Figure 4.5) and slow running queries (Figure 4.6).

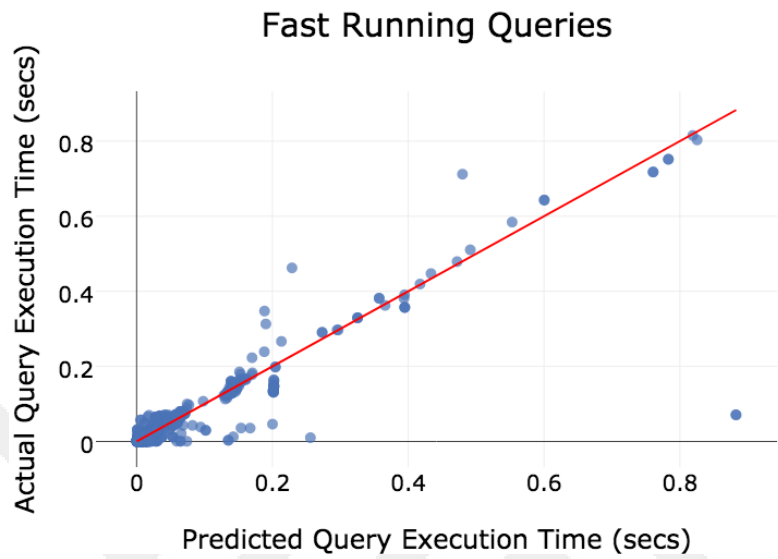

Figure 4.5: Actual vs predicted query execution time in seconds in high workload with query plan features and load features for fast running queries

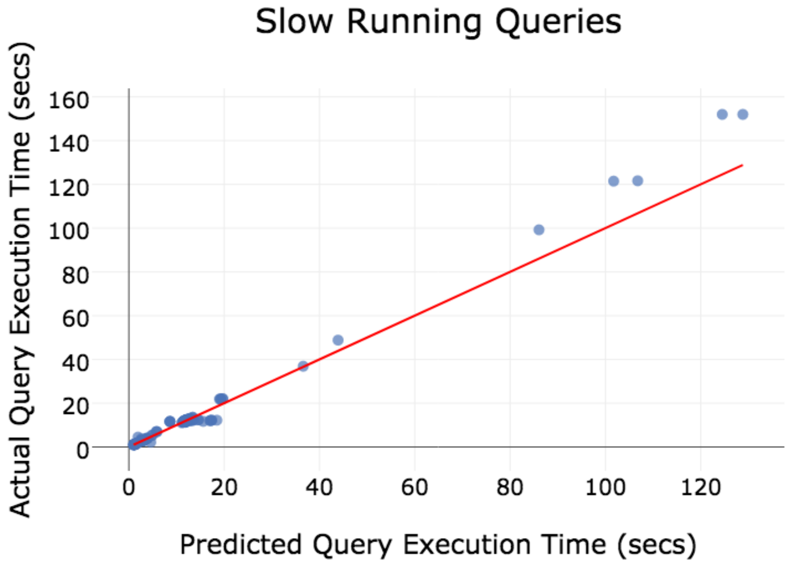

Figure 4.6: Actual vs predicted query execution time in seconds in high workload with query plan features and load features for slow running queries

4.2 Linear Regression Results

We show the prediction results with Linear Regression algorithm for query plan features and load features by using RMSE metric (Figure 4.7).

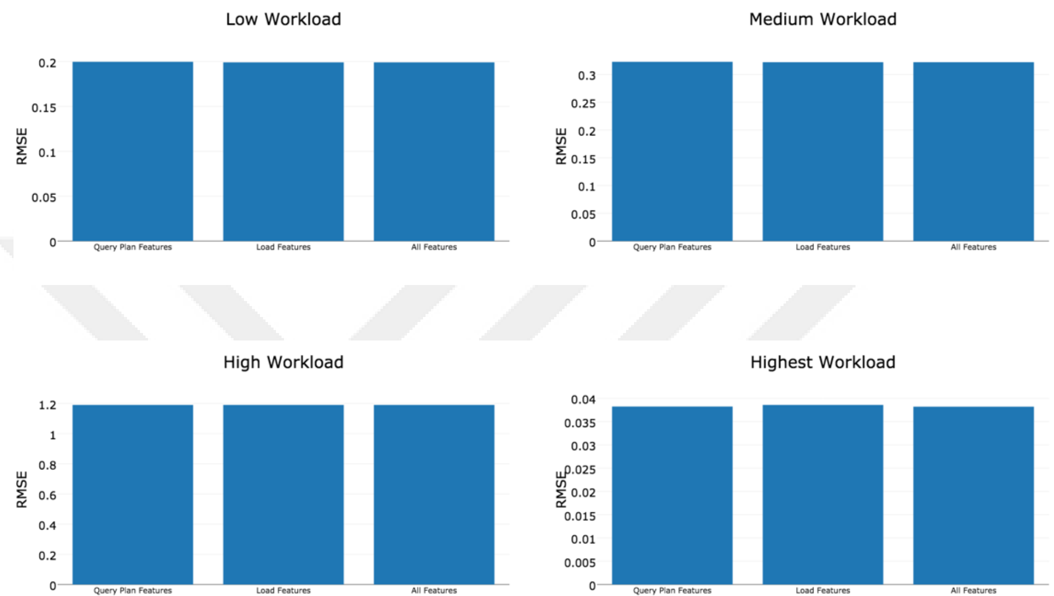

Figure 4.7: RMSE metrics for query plan features, load features and all features comparison in low, medium, high and highest workloads with Linear Regression

In Linear Regression, the predictions are worse than Random Forest Regression by RMSE metrics in all workloads and the feature engineering strategies don't make significant difference. The query execution time predictions with Linear Regression algorithm are as good as Random Forest Algorithm in the highest workload and to interpret this result, we look at standardized coefficient magnitudes of all features in highest workload (Figure 4.8). In Linear Regression of H2O Framework a standardized coefficient is a linear coefficient that has been defined in terms of standard deviations rather than whatever the original units for that particular variable were.

When we interpret the standardized coefficient magnitudes which are used as feature importance, we see that *bytes_in* and *bytes_out* features are the most important feature in the highest workload with Linear Regression model. These two features are related

with the network bandwidth so that in the highest workload, the network bandwidth may be a bottleneck for query execution time prediction. We also see that *table_1_name* is the third most important feature in the highest workload as in the prediction results of DRF algorithm thus both two algorithms show that, *table_1_name* may have a bias in the highest workload for query execution time prediction.

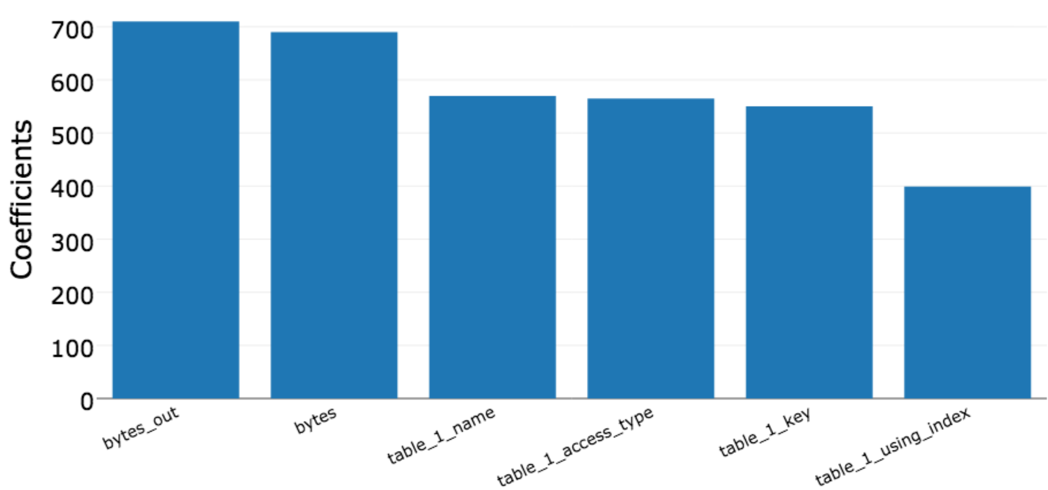

Figure 4.8: Standardized Coefficient Magnitudes in Highest Workload for All Features in Linear Regression Algorithm

5. CONCLUSION

We present load related feature engineering strategies to predict query execution time using machine learning techniques. We make our experiments with already executed queries from a payment service provider in different workloads, introduce load-based features and compare them with traditional query plan features. We show that load based features are effective when they are used with query plan features for both slow and fast running queries in all workload sizes. All these features can be used in machine learning for capacity planning and workload management of autonomous database systems.

Given query plan features and load features in its input feature vectors, the random forest algorithm predicts query execution time and we calculate scaled importance of each feature. The bytes transferred in the current window is one of the most important features in load features and this result shows that the network data transfer volume may allow us to better predict query execution time and to find network bottlenecks on a database server.

Indeed, in high workloads, the workload and also the machine learning algorithms can have a bias to some tables or indexes thus the experiments may depend on the dataset. To solve this bias, we would like to have a random sample which have random distributions of tables and indexes in the future.

In the future, firstly we would like to test multiple sliding window sizes for the aggregations of load features because we believe that the window size should change dynamically by workload size. Second, we plan to extract new load features by using SQL text statistics like the mean and the maximum of aggregation count, equijoin count etc. and compare these features with the query plan features. Third, we plan to apply

these black box load features in different domains and use these features for intelligent workload management of different systems.

We also plan to implement an alert management system to identify long-running queries before they are executed so that we can manage database workload without any downtime. Another direction we plan to research in the future is integrating machine learning models to relational database management systems so that these systems can manage workload more efficient.

REFERENCES

Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., & Zdonik, S. B. (2012, April). Learning-based query performance modeling and prediction. In Data Engineering (ICDE), 2012 IEEE 28th International Conference on (pp. 390-401). IEEE.

Ambati S, et al. “Building Random Forest an Scale” (2014). Published by H2O.ai, Inc.

Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.

Bunea, F., Tsybakov, A. B., & Wegkamp, M. H. (2007). Aggregation for Gaussian regression. The Annals of Statistics, 35(4), 1674-1697.

Chai, T., & Draxler, R. R. (2014). Root mean square error (RMSE) or mean absolute error (MAE)?. Geoscientific Model Development Discussions, 7, 1525-1534.

Chaudhuri, S. (1998, May). An overview of query optimization in relational systems. In Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (pp. 34-43). ACM.

Chawla, N. V., Hall, L. O., Bowyer, K. W., & Kegelmeyer, W. P. (2004). Learning ensembles from bites: A scalable and accurate approach. Journal of Machine Learning Research, 5(Apr), 421-451.

Council, T. P. P. (2008). TPC-H benchmark specification. Published at http://www. tcp. org/hspec. html, 21, 592-603.

Duggan, J., Cetintemel, U., Papaemmanouil, O., & Upfal, E. (2011, June). Performance prediction for concurrent database workloads. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (pp. 337-348). ACM.

Frisch, M. J., Frisch, A., & Foresman, J. B. (1996). Gaussian 94 User's Reference. Gaussian.

Ganapathi, A., Kuno, H., Dayal, U., Wiener, J. L., Fox, A., Jordan, M., & Patterson, D. (2009, March). Predicting multiple metrics for queries: Better decisions enabled by machine learning. In Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on (pp. 592-603). IEEE.

Gupta, C., Mehta, A., & Dayal, U. (2008, June). PQR: Predicting query execution times for autonomous workload management. In Autonomic Computing, 2008. ICAC'08. International Conference on (pp. 13-22). IEEE.

Hamilton, J. D. (1994). Time series analysis (Vol. 2). Princeton: Princeton university press.

Hasan, R., & Gandon, F. (2014, August). A machine learning approach to SPARQL query performance prediction. In Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on (Vol. 1, pp. 266-273). IEEE.

H2O Reference Documentation (2018). Chapter: Architecture.

Lazowska, E. D., Zahorjan, J., Graham, G. S., & Sevcik, K. C. (1984). Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc..

Liaw, A., & Wiener, M. (2002). Classification and regression by randomForest. R news, 2(3), 18-22.

Matsunaga, A., & Fortes, J. A. (2010, May). On the use of machine learning to predict the time and resources consumed by applications. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (pp. 495-504). IEEE Computer Society.

Menze, B. H., Kelm, B. M., Masuch, R., Himmelreich, U., Bachert, P., Petrich, W., & Hamprecht, F. A. (2009). A comparison of random forest and its Gini importance with standard chemometric methods for the feature selection and classification of spectral data. BMC bioinformatics, 10(1), 213.

MySQL version 5.7 Reference Manual (2017), Chapter: Optimization.

Nakagawa, S., & Schielzeth, H. (2013). A general and simple method for obtaining R2 from generalized linear mixed-effects models. Methods in Ecology and Evolution, 4(2), 133-142

Olorisade, B. K., Brereton, P., & Andras, P. (2017). Reproducibility in Machine Learning-Based Studies: An Example of Text Mining.

Picard, R. R., & Cook, R. D. (1984). Cross-validation of regression models. Journal of the American Statistical Association, 79(387), 575-583.

Rasmussen, C. E. (2004). Gaussian processes in machine learning. In Advanced lectures on machine learning (pp. 63-71). Springer, Berlin, Heidelberg.

Seber, G. A., & Lee, A. J. (2012). Linear regression analysis (Vol. 329). John Wiley

Segal, M. R. (2004). Machine learning benchmarks and random forest regression.

Splunk Reference Documentation (2018). Chapter: Streams.

Strobl, C., Boulesteix, A. L., Zeileis, A., & Hothorn, T. (2007). Bias in random forest variable importance measures: Illustrations, sources and a solution. BMC bioinformatics, 8(1), 25.

Willmott, C. J., & Matsuura, K. (2005). Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. Climate research, 30(1), 79-82.

Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., & Naughton, J. F. (2013, April). Predicting query execution time: Are optimizer cost models really unusable?. In Data Engineering (ICDE), 2013 IEEE 29th International Conference on (pp. 1081-1092). IEEE.

Wu, W., Chi, Y., Hacígümüş, H., & Naughton, J. F. (2013). Towards predicting query execution time for concurrent and dynamic database workloads. Proceedings of the VLDB Endowment, 6(10), 925-936.

Wu, W., Wu, X., Hacigümüş, H., & Naughton, J. F. (2014). Uncertainty aware query execution time prediction. Proceedings of the VLDB Endowment, 7(14), 1857-1868.

BIOGRAPHICAL SKETCH

Yalçın Yenigün, born in 1986 in Balıkesir/Turkey, is a software engineer and data scientist specializing in machine learning with 9 years of experience in the full lifecycle of the software design process including requirements definition, prototyping, proof of concept, design, interface implementation, testing and maintenance.

Education

- M.S., Computer Engineering, Galatasaray University, 2018 (expected).
- B.S., Computer Engineering, Galatasaray University, 2010.
- High School, Mathematics, Sırrı Yırcalı Anatolian High School, 2004

Work Experience

2015 – present, Expert Software Engineer, iyzico.
2013 – 2015, Unit Manager, Bilge Adam
2010 – 2013, Software Development Engineer, Vodafone.
2009 – 2010, Software Development Specialist, Zerobuffer.

Publications

- Öztürk, M., Yenigün, Y., Yaycıoğlu, O. T., & Tunalı, A. Ç. (2012). RESTful Konfigürasyon Yönetimi Web Arayüzü. 6. Ulusal Yazılım Mühendisliği Sempozyumu
- Yenigün, Y., Ozgovde, A., Dincel, O. (2018). Load Related Feature Engineering For Query Execution Time Prediction. 22nd European Conference on Advances in Databases and Information Systems. (submitted)