International Journal of Applied Engineering Research ISSN 0973-4562 Volume 13, Number 19 (2018) pp. 14298-14302

Algorithmic Analysis of Execution Plans – Using different algorithms to find the least performing queries

Sharad Narayan Sharma, Akshar Dhondiyal
Department of Computer Science and Engineering, Vellore Institute of Technology, Katpadi Road, Vellore, Tamil Nadu-632014, India
Department of Computer Computer Science Engineering, Motilal Nehru National Institute of Technology, Barrister Mullah Colony,Teliarganj, Allahabad, Uttar Pradesh – 211004, India
Email id: sharmasharad795@gmail.com, akdhondi@gmail.com

Abstract

Data can be stored in different kind of databases. A few common ones include NoSQL (Not Only SQL) databases, Big Data oriented databases such as MongoDB and perhaps the most commonly used: relational database. One of the reasons for its continuous popularity is in its ability to store structured data and in its capability to easily retrieve the stored data. We have worked with Oracle relational database in this paper. Based on the known condition and criteria, one can use Structured Query Language (SQL) to extract data from Oracle databases. There are various types of SQL statements. But our focus will be on Data Manipulation Language (DML) Statements, which include the SELECT, INSERT, UPDATE and DELETE statements.

The execution of DML statements brings the Oracle optimizer into action. It then chooses particular execution plans for each such statement. A distinct value (numeric) is created by the optimizer whenever it generates an execution plan for a DML statement. This value is called the PLAN_HASH_VALUE. These values are stored in the DBA_HIST_SQLSTAT view. Our primary goal is to use the parameter values associated with different PLAN_HASH_VALUES in this view along and find out those SQLs which are highly likely to fail or which might affect the performance of the database as a whole, due to their poor performance: 'problematic SQLs'. We will use different algorithms and data segregation methods for this procedure, check which SQL queries are highly likely to fail in the future and find which algorithms or set of algorithms might be the best for the particular problem. We will use Python and its libraries for this purpose. Additionally, we shall then apply k-nearest neighbours algorithm (KNN) and Support Vector Machines (SVM) to the final dataset and find out the accuracy of the model. We will also represent these results graphically and experiment with different parameter values to find the ideal model. Hence, our approach will use algorithms on different parameters of execution plans and use machine learning algorithms on those modified datasets.

Keywords— Oracle database, Execution plans, PLAN_HASH_VALUE, DBA_HIST_SQLSTAT, Support Vector Machine

INTRODUCTION

An extremely complex database environment can support more than a billion queries per day. All it takes is one SQL performing poorly to tip things over the edge. We often notice that using the same query for extremely different data volume scenarios causes unexpected performance issues. If an SQL is trying to do two or more very different things, we should design for it rather than hoping that the optimizer shall handle it. In this paper, we suggest our approaches to handle such a problem.

A particular Oracle database user can view the execution plans chosen by the Oracle Optimizer using the EXPLAIN PLAN statement. An execution plan is nothing but a step of operations which Oracle needs to perform in order to run the SQL statement. The execution plans can vary with variance in the optimizer inputs.

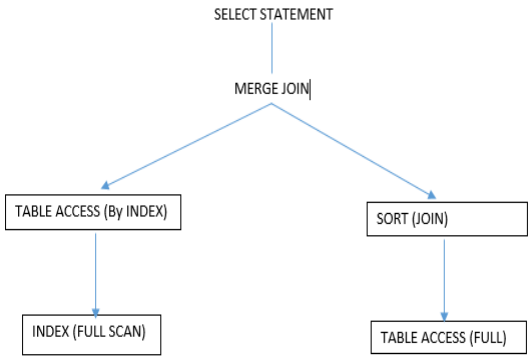

Figure 1. How the Optimizer chooses different execution plans

This results in multiple execution plans for the same SQL query. More generally, execution plans differ due to differences in the execution setup. Execution in different schemas and also different costs are the two primary reasons for this. However, the execution plan alone cannot be used to distinguish between poorly performing statements and those that perform well. In this paper, we are going to use four different algorithms to solve this problem. We're going to borrow data stored in a view called DBA_HIST_SQLSTAT for this purpose. This view usually stores historical information about SQL statements. It includes the top SQL statements based on a set of rules. It leverages its information from V$SQL as well. The DBA_HIST_SQLSTAT view is often used with DBA_HIST_OPTIMIZER_ENV,

DBA_HIST_SQLTEXT, DBA_HIST_SQL_PLAN to provide a complete historical statistical picture.

For our cause, we shall only use the SQL_ID, PLAN_HASH_VALUE, EXECUTIONS_DELTA, ELAPSED_TIME_DELTA, BUFFER_GETS_DELTA, CPU_TIME_DELTA parameters of the DBA_HIST_SQLSTAT view. We will perform our algorithms on these parameters. We have chosen the delta values over the total values because we are interested in finding the current query state attributes. The delta values are ideal since they represent the values of the query between the prior snap and the current snap. Since the operations have been performed on data which is confidential, we will be creating a small mock dataset for visual and explanation purposes.

Knowledge of how SQL queries work is very important in order to analyse the values in the DBA_HIST_SQLSTAT view. A couple of Python libraries, Pandas and Numpy are used for the data analysis and implementation of algorithms. Our Support Vector Machine model which is used on the final dataset obtained will feature the Gaussian prediction model. We have leveraged the scikit – learn library for this purpose along with matplotlib library to create a visual interpretation of the models created.

LITERATURE REVIEW

Jean Habimana provides solutions which can work as a reference point for developers while tuning the overall performance of a database. His paper has methods which can help in the development and maintenance of the queries used to extract data from the database. He mentions that query optimization plays an important role in the performance of a database and the optimization keeps evolving with better and complex optimization approaches. Jean records the average time associated with each query. He shows that the speed increases when the more efficient query is used. He introduces ten tips in order to optimize the query. His tips or strategies include using column names instead of * in SELECT queries, avoiding a HAVING clause in SELECT queries, eliminating DISTINCT conditions which are not necessary, avoiding NEST conditions in subqueries, using an IN predicate with indexed columns, avoiding OR conditions in join conditions, using UNION ALL instead of UNION, removing redundant mathematical operations in the query. The data obtained by these refined queries can used by our algorithms for even better query failure prediction. The algorithms can be used on top of refined and unrefined queries to see if the number of queries having a probability of failure or causing problems in the database, has actually decreased or not once the refined queries are used to extract the dataset.

Christopher Koch discusses about using a XML-based Execution Plan Format (XEP). XEP has been described as a simple readable format for SQL execution plans. XEP is also not heavy-weight and can be easily exchanged. This is because it is built on top of XML technology, that is, the framework of XEP is made using XML language for the schema. Koch discusses about the two strategies he takes for XEP representation of DBMS related execution plans. These approaches have been evaluated on IBM DB2 DBMS, Oracle Database and Microsoft SQL. He says that plan information and execution plans are handled on an abstract level by XEP. A XEP executionPlan object is used to get the information and XEP operator objects are used to capture the operators related to the plan. For conversion of DBMS related execution plans into XEP format, Koch uses a transformer of execution plans. The most common DBMS such as Oracle Database, Microsoft SQL Server and PostgreSQL have the ability to export execution plans into XEP format. For such cases, Koch develops an Extensible Stylesheet Language Transformation (XSLT). For DBMSs such as MySQL, IBM D2 LUW and IBM DB2 z/OS which do not provide support for XML plan output, Koch uses a Java application based approach. Our algorithms developed in this paper can be used on top of the XEP derived dataset from the Oracle database and check the variance in results compared to non -XEP derived data. The SVM model could further test the models' accuracy built by these two approaches, for Oracle database.

OUR CONTRIBUTION

In this section, we will describe how we got to our dataset and our final application of algorithms. After this, we will discuss about our SVM and KNN models. As mentioned before, we'll be attaching mock datasets of the actual data that we have used at each step, due to confidentiality purposes. First, we build a query to extract the preliminary dataset from the DBA_HIST_SQLSTAT view.

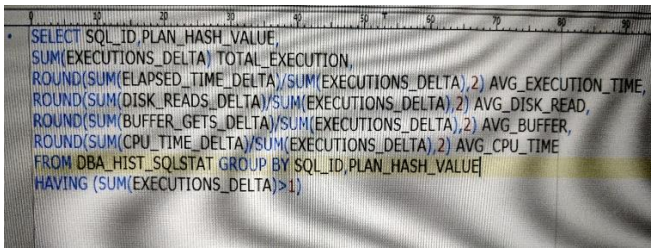

Figure 2. SQL Query used to get the first dataset

In this query, we have grouped the data by SQL_ID and PLAN_HASH_VALUE to get the executions plans and its parameter values associated with each query. We are interested in finding the total number of executions for each plan and each plan's average execution time (microseconds), average number of disk reads, average number of buffer gets and the average CPU time(microseconds). These average values are associated with the delta values, not the cumulative ones. After getting this dataset, we remove those SQL_IDs from consideration which only have a single PLAN_HASH_VALUE. We only look for those queries which took different paths of execution for the same query and hence, might have a scope for failing.

We perform our algorithms on the above obtained dataset. We import Pandas and Numpy libraries from Python for this task. We shall now discuss our algorithms one by one and our reasons for choosing them.

SQL_ID	PLAN_HASH_VALUE	TOTAL_EXECUTIONS	AVG_EXECUTION_TIME	AVG_DISK_READ	AVG_BUFFER_GETS	AVG_CPU_TIME
ABC	2345	5871	12.43	0	3.74	41.07
CDE	132343534	41	41.21	7.8	2198.7	993.78
EDF	567897898	27	17149.76	6.48	11.4	13456.9
ERF	34543535	152995	283.31	127.25	2205.98	73512
POU	7688798	355	41.21	3774.89	59.76	6076.34

Figure 3. Dataset that we get from the SQL query (Mock dataset)

The first algorithm leverages AVG_EXECUTION_TIME and AVG_CPU_TIME. It is based on the concept that queries that have a longer average execution time and a shorter average CPU time are more likely to cause problems. This is because such queries will be spending a larger portion of their execution time not utilising the CPU. This might cause a longer waiting time for other queries in the queue.

Hence, we are first grouping the queries by SQL_ID and performing a subtraction operation between the average of all AVG_EXECUTION_TIME and average of all AVG_CPU_TIME values associated with the multiple PLAN_HASH_VALUEs of a SQL_ID. The larger this difference, the more chances of a SQL to be problematic.

```
sql1 = sql.groupby('SQL_ID')
ET_minus_cp_time =
sql1['AVG_EXECUTION_TIME'].agg(np.mean)
– sql1['AVG_CPU_TIME'].agg(np.mean)
```

The second algorithm visits the worst case scenario of the first algorithm. It is based on similar reasons as the first. First, grouping is done based on the SQL_ID. But here, we then take the difference between the maximum value of AVG_EXECUTION_TIME and minimum value of AVG_CPU_TIME associated with the multiple PLAN_HASH_VALUEs of a SQL_ID.

```
ET_minus_cp_time_worst_case =
sql1['AVG_EXECUTION_TIME'].agg(np.mean)
– sql1['AVG_CPU_TIME'].agg(np.min)
```

The third algorithm is based on the idea of variance, that is, how far the values are from the mean of a parameter. This will help us tell which SQL is more likely to have extreme parameter values associated with PLAN_HASH_VALUE, either too small or too big. In either case, this shall increase the value of the variance, and hence, indicate a higher chance of the SQL failing. This is because a higher variance corresponds to a higher fluctuation in parameter values, which means that the optimizer can choose two very different kinds of execution plans for the very same SQL.

```
stdev_ettime=
sql1['AVG_EXECUTION_TIME'].agg(np.std)
stdev_diskread=
sql1['AVG_DISK_READ'].agg(np.std)
stdev_buffer_get=
sql1['AVG_BUFFER'].agg(np.std)
stdev_cpu_time=
sql1['AVG_CPU_TIME'].agg(np.std)
```

The fourth algorithm is based on finding the difference between the maximum and the least values of a parameter for a SQL_ID for all its PLAN_HASH_VALUEs. For example, let us take the example of AVG_EXECUTION_TIME for the various executions plans of a query. Our initial idea was to find the difference between its minimum and maximum values, and then divide this value by the minimum. However, such an algorithm often ignores the impact of all the plans that were generated by the optimizer. This has been explained in the figure 4. Both the cases have the same value according to the initial algorithm, even though it can be clearly seen that the execution time values are higher for the second case. We thought it was imperative to include the values that were not getting incorporated into the (max – min)/min formula, to get to better understanding of a problematic SQL. Hence, we modified the algorithm and multiplied the value obtained from the initial algorithm by the mean.

```
max_min_ettime =
sql1['AVG_EXECUTION_TIME'].agg(np.max) –
sql1['AVG_EXECUTION_TIME'].agg(np.min)

max_min_ettime = max_min_ettime /
sql1['AVG_EXECUTION_TIME'].agg(np.min)

max_min_ettime = max_min_ettime *
sql1['AVG_EXECUTION_TIME'].agg(np.mean)
```

	Case 1	Case 2
	1	100
	4	99
	2	34
	3	125
	5	25
Max-Min/Min	(5-1)/1 = 4	(125-25)/25=4
New algo	4*3 = 12	4*76=306.4

Figure 4. Two test cases for the fourth algorithm

We apply similar algorithms for AVG_CPU_TIME, AVG_DISK_READ and AVG_BUFFER.

We put the results of all these algorithms into another excel. We ignore the values under buffer gets and disk reads for the fourth algorithm because a large number of values were tending towards infinity due to a high number of 'minimum' values being 0. All these values under the different algorithms were then sorted in descending order to get the worst performing algorithms on top of our results.

Now, in order to apply the SVM and KNN algorithms, we carry out a little bit of data arrangement. For each algorithm, we pick out the worst 10 for each of them and remove the duplicates from this cumulative list.

We got a total of 23 values after this operation. Now all these SQL_IDs are assigned a value of 1 while the others are given a value of 0. A value of '1' corresponds to that SQL having a higher chance of failing or causing problems while a value of '0' suggests a well-tuned SQL, with lesser chances of failing.

SQL_ID	Failure Vali	ALGO1	ALGO2	ALGO3_STDETTIME	ALGO3_STDDISKR	ALGO3_STDCPU	ALGO3_STDBUFF	algo4_et_time	algo4_cpu_time
ABC	1	324927.2	790262	198.23	1115387.39	371795.7967	389291.1892	2027.34	371795.8
CDE	0	18.245	287.64	18537.12	18843.005	6281.001667	8385.679268	43901.23	8385.679
EDF	0	1550.91	77027.7	144082.87	222661.48	74220.49333	74561.76184	13093.3	74561.76
ERF	1	37.25	120443.2	102.11	120582.6	40194.2	54460.4455	12.4	40194.2
POU	0	863034.9	9619.98	115798.76	988453.6	329484.5333	394979.2597	6753323	394979.3

Figure 5. Dataset with the algorithm values and the 'failure values' assigned to each SQL_ID (Mock Dataset)

We distributed the data in the same manner for both SVM and KNN. The column with 1s and 0s values mentioned above, was the 'y' or the dependent SQL_ID value for the dataset. For the independent values 'X', we experimented with multiple test cases for both SVM and KNN. About 10% of the dataset has been set aside as the test set and the rest as training set for KNN, while this figure is 20% for SVM. We observed that a smaller test set gave better results as our dataset itself was not huge after all the mentioned operation and algorithms where performed on it.

For KNN, we have used the Euclidean distance between the points while implementing the algorithm. We observe that the model was the most accurate when the K value or the n-neighbours parameter of the KNeighborsClassifier object was set to 3. The test set accuracy becomes stagnant after K value increases from 3 onwards when all algorithms are included. The combination of algorithms 1 and 2 gives a surprising 100% accuracy on the test model.

K Value	Independent X	Train Set Accuracy	Test Set Accuracy
3	All algorithm values (ARV)	93.50%	88.89%
4	ARV	90.9%	77.78%
5	ARV	90.9%	77.78%
6	ARV	87.01%	77.78%
7	ARV	88.3%	77.78%
3	Algorithm 1 &2	92.21%	100%
8	Algorithm 1 &2	85.71%	88.89%
3	Algo3_STDBUFFER & ALGO4_ET_TIME	88.32%	88.89%

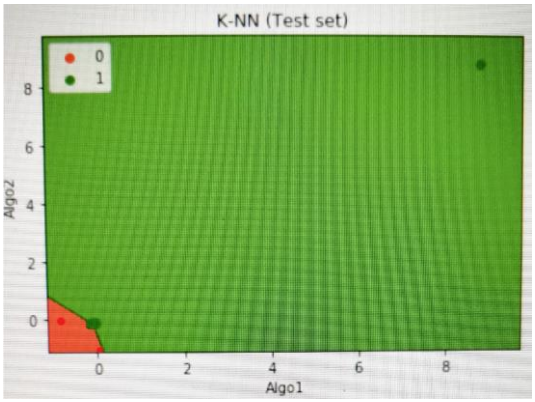

Figure 6. Graphical representation of the KNN model with algorithm 1 and 2

SUMMARY AND CONCLUSION

We leveraged data from DBA_HIST_SQLSTAT table to help us arrive at four algorithms. These algorithms were then used to predict which SQL queries can be troublesome or a failure in the future.

As can be seen from the figure 6, the range of values in algorithms 1 and 2 for which a SQL query will perform well is a small one.

For SVM, we have used the Gaussian kernel to get a better graphical representation of our model. We have experimented with different algorithm combination for this model, a few of which have been displayed in the table. We observe that this model gives the highest accuracy when the independent variable involved in combination with other algorithms, is algorithm 2.

Independent X	Train Set Accuracy	Test Set Accuracy
All algorithm values	89.7%	83.33%
Algorithm 1 & 2	82.35%	83.33%
ALGO3_STDBUFFER & ALGO4_ET_TIME	92.64%	83.33%
Algorithm 2 & ALGO3_STDETTIME	83.82%	83.33%
ALGO3_STDDISKREAD & ALGO3_STDCPU	79.41%	72.22%
ALGO3_STDDISKREAD & ALGO4_CPU_TIME	76.47%	66.67%

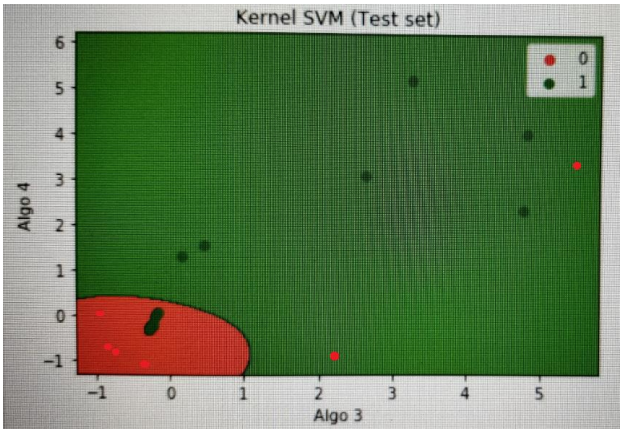

Figure 7. Graphical representation of the SVM model with ALGO3_STDBUFFER and ALGO4_ET_TIME

- The four algorithms gave different results for the worst performing queries.
- The best way to utilize these algorithms is to use them in combination with each other.
- One can trace back to the actual queries of the problematic SQLs returned by our algorithm and then fine tune them.
- We have tried to use algorithms which incorporate each vital parameter associated with a SQL.
- The accuracy of these algorithms will increase with larger datasets.
- The fact that a SQL might fail is not just related to its number of execution plans but to the parameter values associated with these plans.
- We tried different KNN and SVM models, out of which KNN gave more accurate models.
- KNN gave very highly accurate models when only algorithms 1 and 2 were taken into account.
- SVM models gave the highest accuracy of 83% and tended to perform well when values of algorithm 2 were used as the independent variable.

REFERENCES

[1] Jean Habimana (2015, October). Query Optimization Techniques - Tips For Writing Efficient And Faster SQL Queries. In International Journal of Scientific & Technology Research Volume 4, Issue 10

[2] Christoph Koch (2016). XML-based Execution Plan Format (XEP).In Open Journal of Databases (OJDB) Volume 3, Issue 1

[3] DBA_HIST_SQLSTATOracle Database Reference 10g Release 1 (10.1) Part Number B10755-01

https://docs.oracle.com/cd/B13789_01/server.101/b10755/statviews_2158.htm

[4] Database Performance Tuning Guide Oracle Database Online Documentation, 10*g* Release 2 (10.2) / Administration

https://docs.oracle.com/cd/B19306_01/server.102/b14211/ex_plan.htm#g42231