AutoDI: Towards an Automatic Plan Regression Analysis

Hai Lan*, Yuanjia Zhang†, Zhifeng Bao*, Yu Dong†, Dongxu Huang†, Liu Tang†, Jian Zhang†
*RMIT University, †PingCAP
{hai.lan,zhifeng.bao}@rmit.edu.au,{zhangyuanjia,huang,tl,yu.dong,zhangjian}@pingcap.com

ABSTRACT

Manual analysis on plan regression is both labor-intensive and inefficient for a large query plan and numerous queries. In this paper, we demonstrate AutoDI, an automatic detection and inference tool that has been developed to investigate why a sub-optimal plan is obtained by analyzing two different plans of the same query. AutoDI consists of two main modules, *Difference Finder* and *Inference*. The former aims to find where the two plans are different, and the latter tries to obtain the reasons why the differences come out. In our demonstration, we use a real plan regression in TiDB to show how AutoDI works.

PVLDB Reference Format:
Hai Lan, Yuanjia Zhang, Zhifeng Bao, Yu Dong, Dongxu Huang, Liu Tang, Jian Zhang. AutoDI: Towards an Automatic Plan Regression Analysis. PVLDB, 15(12): 3626-3629, 2022.
doi:10.14778/3554821.3554860

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rmitbggroup/AutoDI.

1 INTRODUCTION

Motivation. Database vendors have been spending substantial resources on plan quality assurance. There are two common ways to achieve so. The first way is via a SQL tuning advisor [3, 4], targeting for database users to avoid a sub-optimal plan on a production database system. For example, Oracle provides the SQL Tuning Advisor [3], where users can submit one or more SQL statements as the input to the advisor and receive advice or recommendations on how to tune the statements, e.g., updating statistics, along with a rationale and expected benefit. It tries the possible optimizations, e.g., calling the index advisor to create or drop some indexes. By comparing the performance of the default plan with the newly generated plan that has a lower query latency, it gives the recommendations. Obviously, it is uninformed for optimizer developers to quickly find the reason why such differences (performance regression or performance improvement) come out. The second way is via the optimizer testing [5, 8, 10], targeting for the optimizer development team before releasing a new optimizer version. Most of these studies introduce a metric to assess the accuracy of the optimizer based on the estimated *cost rank* and *runtime rank* of the possible plans. These metrics can be an early warning of an optimizer while they still could not provide the sufficient feedback from a certain query.

A simple but important question is: ***if we find that a worse plan was chosen by the optimizer in default during the testing, what are the reasons behind that?*** Two common scenarios that sub-optimal plans are generated can happen in the optimizer test:
Case 1: Different Cost Rank and Runtime Rank. For each query, a plan enumeration process will be called to find a set of plan candidates that are equivalent to the one generated in default. These plans will be costed and executed. After that, a plan regression will be found if a plan with larger cost than the default one has a lower runtime.
Case 2: Regression In Different Optimizer Versions. The newly generated plan will be also compared to the one generated by the last optimizer version. Similarly, a plan regression will be found if the runtime of the new plan is larger than the old one.

To answer the above question, a manual analysis has to be conducted by the optimizer development team at the moment. Such an analysis consists of two steps: 1) find the differences between two plans, 2) check out possible reasons with the corresponding collected data e.g., operators' cardinality, estimated cost, and running time. However, if the number of problematic queries or the plan tree is large, it becomes *labor-intensive* work and is challenging for a developer to quickly find out the differences between two different plans, and more importantly, the underlying reasons for OLAP queries.
Studied Problem. Given two different plans of the same query, P_1 and P_2, P_1.cost < P_2.cost while P_1.runtime > P_2.runtime, report the differences between different plans and the reasons behind them.
Challenges & Our Solution. There are three challenges in building an automatic tool to solve the above problem:
Challenge 1. To find out the parts (operator or subtree) of two plans with different implementations, they must have the same semantics. In Figure 1, the 'Sort' node in P_2 is added during the optimization to satisfy the order requirement of 'MergeJoin' while it is not in 'HashJoin' of P_1. Different from P_1, there is no explicit 'Sort' operation on *t1.a* in P_2. We need to find out which operation in P_2 is equivalent to it.
Challenge 2. If multiple differences exist between two plans, the proposed tool should be able to figure out which one is more important, such that it give a more *actionable* report for the developers. In Figure 1, there are two differences, i.e., the join operator and the data access method on *t1*. We need to know which one is the main reason that results in different runtime.
Challenge 3. The proposed tool should be *extensible*. With the development of an optimizer, new difference type, possible reasons, and operators could be introduced. The proposed methods must conveniently support the new features as well.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554860

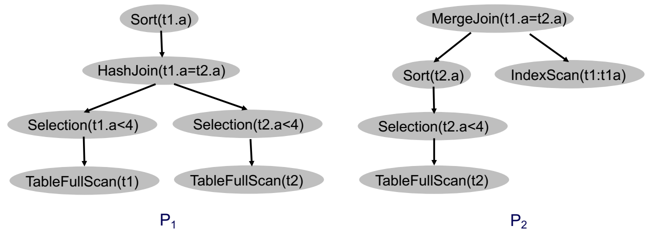


Figure 1: Two different plans of the query *select * from t1 join t2 where t1.a=t2.a and t1.a < 4 order by t1.a*. An index *t1a* is built on *t1*'s attribute *a*.

To this end, we have developed an automatic detection and inference tool, AutoDI, to help the optimizer development team quickly find the reasons why a sub-optimal plan is generated. Although we demonstrate AutoDI on TiDB [6], an open source distributed HTAP database, it is generally applicable to other databases with the trivial efforts on extension, e.g., introducing their own operator notations, collecting the plans from their APIs. Inspired by the manual analysis process, AutoDI consists of two main modules, *Difference Finder* and *Inference*. The former finds where the two plans are different, and the latter tries to obtain the reasons why the differences come out. To our best knowledge, this is the first attempt to help optimizer developers analyze the reasons of plan regression and we hope it can attract more community efforts on this topic to enhance automatic detection and analysis.

2 RELATED WORK

Optimizer Analysis & Test. Leis et al. [7] study the impact of three optimizer modules, i.e., cardinality estimation, cost model, and plan enumeration algorithm on selecting an optimal plan. They find that the errors in cardinality estimation are usually the reason for bad plans. Perron et al. [9] conduct a similar study and propose a re-optimization method to improve the accuracy of cardinality estimation. In [5, 8, 10], the authors try to propose a new metric as an indicator of optimizer quality for a workload during the optimizer test. All these metrics are based on the estimated cost rank and runtime rank of the possible plans.

Related Tools. Many database vendors have their own tools for users to avoid plan regression or a sub-optimal plan [1–4, 11] in a production system. For example, Oracle provides the SQL Tuning Advisor [3], where users can submit one or more SQL statements as input to the advisor and receive advice or recommendations for how to tune the statements, e.g., updating statistics, building indexes, along with a rationale and expected benefit. SQL Server provides *Compare Showplan* [2] to highlight the different subtrees of two plans with the same color. Users can further click the operator to obtain the detailed comparisons, e.g., cardinality of the operator and runtime. However, it does not give any analysis on the reasons why the differences comes out.

Difference. Different from [7], AutoDI tries to dig out main reasons for a certain query. Furturemore, AutoDI points out which operators are problematic and the possible reasons on them. In contrast to the above tools targeted for database users, AutoDI targets for the optimizer developers to investigate the reasons why a plan regression happens. It significantly relieves their labor-intensive work on discovering the reasons behind the sub-optimal plans.

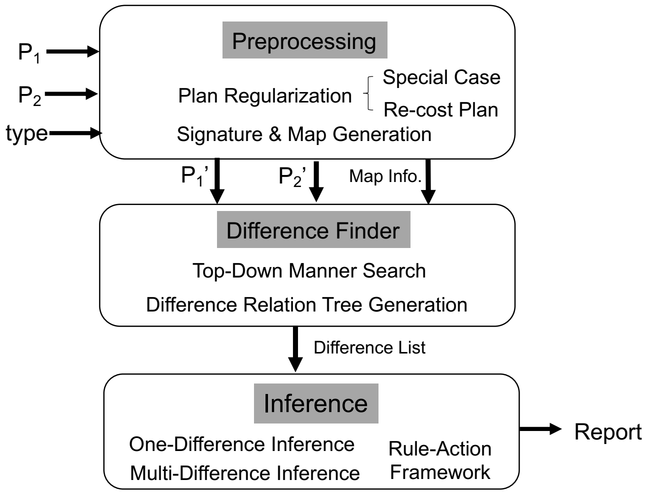


Figure 2: An Overview of AutoDI

3 BACKGROUND AND TOOL DESIGN

3.1 Background

3.1.1 Existing Differences. After investigating several database systems, we find there are four typical types of differences: 1) different data access paths (*DADiff*), e.g., full table scan vs. index scan; 2) different operator implementations (*ImpDiff*), e.g., hash join vs. merge join; 3) different join orders (*JODiff*), e.g., $A \bowtie B \bowtie C$ vs. $A \bowtie C \bowtie B$; 4) different operator orders (*OODiff*), e.g., whether union is under join. *OODiff* rarely occurs and is mainly determined by the logical optimization.

3.1.2 Possible Reasons. We can classify the possible reasons into two categories: 1) Code-level of the optimizer, which includes statistics model, cardinality estimation module, cost model, plan enumeration algorithm, and logical operator implementations. 2) The optimizer's input parameters, such as the used hint set and the built indexes. Most database systems have provided the idea of hint for users to specify the optimizer's behaviors.

3.2 Tool Design

The overview of AutoDI is presented in Figure 2, which mainly consists of three modules: *Preprocessing*, *Difference Finder*, and *Inference*. AutoDI takes two different plans of the same query, and the case type as the inputs, and it outputs a report about the reasons behind their differences.

3.2.1 Preprocessing. To locate the differences between two plans effectively and efficiently, we introduce the preprocessing module to regularize two plans and generate the nodes' signatures. *Plan Regularization* will handle two cases. If the plan regression happened in different optimizer versions, we re-cost and run the plan from an old version in the new one to make the two plans comparable and construct a new test pair. For special case, e.g., in Figure 1, the 'Sort' node can be added with an enforced order property or implicitly implemented by other operators, e.g., 'Index Scan'. The parent node will absorb the added nodes. In Figure 1, 'MergeJoin' will absorb 'Sort', and then two join nodes can be compared.

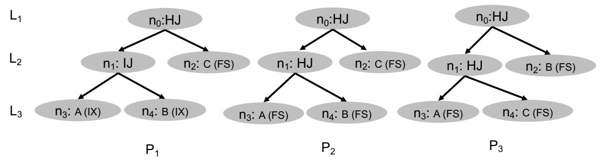


Figure 3: Three Different Plans of One Query. HJ, IJ, IX and FS are hash join, index Join, index scan, and full table scan respectively.

To quickly find the differences, we introduce two signature types, *logical signature* and *physical signature*. If two nodes have the same logical signature, their outputs (the sub-plans with the two nodes as roots) must be the same. If two nodes have the same physical signature, their implementations (the sub-plans with the two nodes as roots) must be the same. We also build a signature map for the second plan of AutoDI input to quickly locate the node for a given logical signature.

3.2.2 Difference Finder. With the plans annotated with logical signatures and physical signatures, *Difference Finder* tries to find the parts where the logical signature is the same while the physical signature is different.

Here, we design a top-down manner algorithm. Each time, we analyze a node pair from two plans. We first check the logical signature. If the logical signatures of two nodes are the same, we then check their physical signatures. If the physical signatures are still the same, we know these two sub-plans are the same and do not need to analyze the descendant nodes. If the physical signatures are different, we further check whether the implementations of two nodes are different or the difference comes from their descendant nodes, and report the differences for the latter. However, if the logical signatures are different, we can conclude the difference is *JODiff*. Then, we use signature map to find the next node pair with the same logical signatures to analyze.

Example 3.1. We use the plans P_1-P_3 in Figure 3 to show our search process. Let n denote a node in the plan tree, $n.ls$ be the logical signature of n, $n.ps$ be the physical signature of n. $n.nt$ and $n.op$ are the node type and node physical operator respectively. In the first case, we try to find the differences between P_1 and P_2:
1) We first compare $n_0.ls$ in P_1 with $n_0.ls$ in P_2, and $n_0.ps$ in P_1 with $n_0.ps$ in P_2. We find that logical signatures are the same while physical signatures are different.
2) We check whether $n_0.nt$ and $n_0.op$ are the same in these two plans. In this case, they are both join and hash join. We can infer that the different parts of these two plans must appear in the subtrees.
3) We compare n_1 in P$_1$ and P_2. Same as n_0, logical signatures are the same while physical signatures are different.
4) When checking $n_1.nt$ and $n_1.op$, we find that they have different implementations and report *ImpDiff*.
5) We continue to check the children of n_1, because other differences may exist in the subtrees, e.g., the data access path, which is one of the differences presented in Sec. 3.1. Also, we find both data access paths on table A and table B are different and report *DADiff*.
6) After visiting all nodes in the left child of n_0, we conduct the same process on its right child, n_2. They are the same in both plans.

When comparing P_1 and P_3, we follow the same steps above, and find out that the logical signatures of n_1 (also shown in n_2) in P_1 and P_3 are different which means they choose a different join order, and hence we report *JODiff*. After the different logical signature, i.e., join order (mostly), is found from one node, we use the logical signature map of P_3 to find the candidate nodes with same logical signature of nodes under n_0 in P$_1$. Here, we quickly locate the leaf nodes of the plan and report the different access paths on table A and B, i.e., two *DADiff* s.

```
Rule {
    /* rule id */
    ruleId
    /* possible reasons to be checked */
    actionList
    /* reasoning flag, 0 or 1. if it is 1, the
corresponding action will be executed  */
    actionFlag
}

Rule4ImpDiff {
    ruleId: "ImpDiff"
    actionList: {ce4imp, cm4imp}
    actionFlag: {1, 1}
}

/* check cardinality estimation */
func ce4imp(node1, node2) {…}
/* check cost model */
func cm4imp(node1, node2) {…}
```

Figure 4: The Rule-Action Framework and an Example of ImpDiff

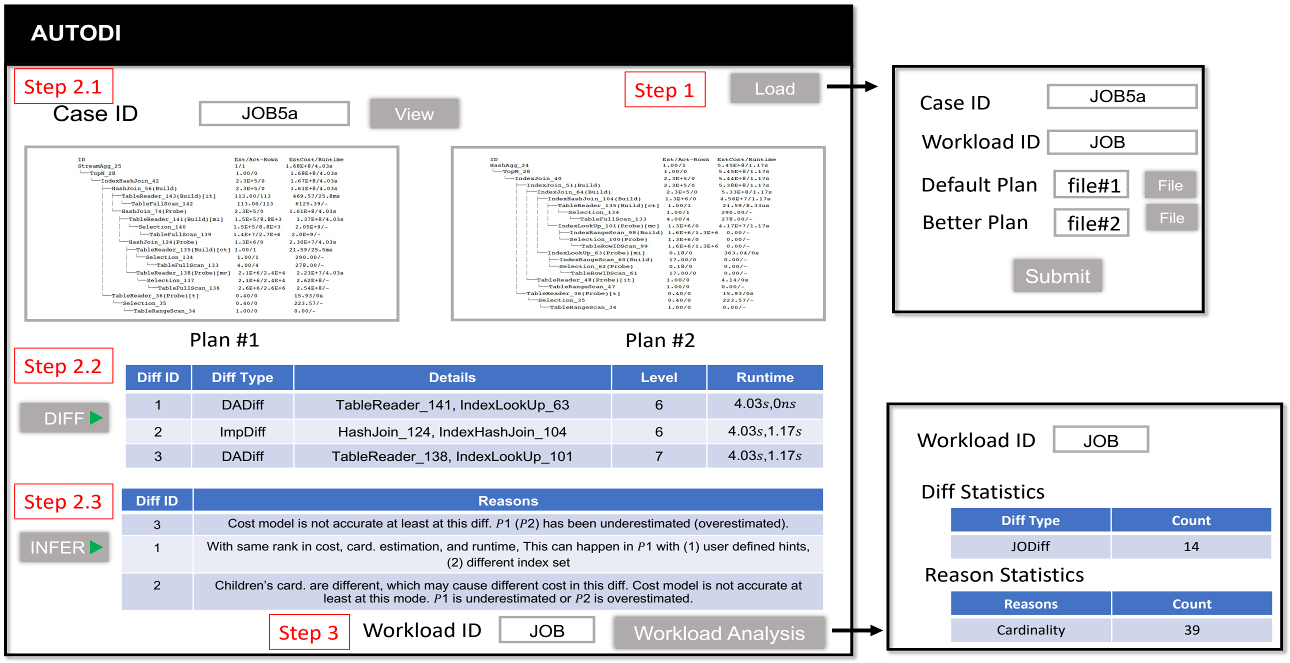
3.2.3 Inference. This module aims to obtain the possible reasons behind the differences found. We introduce a rule-action framework to make it extensible, e.g., for a new difference, which is shown in Figure 4. Each difference type has a corresponding rule. In each rule, the *actionList* includes a list of actions for the possible reasons, and *actionFlag* is a 0/1 array. We execute the i^{th} action if the i^{th} value in *actionFlag* is 1. When a new difference type is introduced, we only need to define a new rule and implement its related actions. When a new possible reason for a difference type comes, we just need to add a new action in its *actionList* and set its flag in *actionFlag*. For each difference, we call the actions for it to obtain the reason.

Ranking of Multiple Differences. We analyze some plan regression cases and make two observations. 1) In most cases, there are few main differences, i.e., the runtime gap between the nodes is close to the runtime gap of two plans. 2) The nodes in the most crucial difference have a higher level than the nodes in other differences. Motivated by the above observations, we introduce a parameter α to define the crucial differences. Suppose two different plans of one query are P_1 and P_2, where $P_1.rt$ is smaller than $P_2.rt$. $P.rt$ indicates the runtime of plan P. A difference *diff* could be a crucial difference if *(diff.*n_1*.rt-diff.*n_2*.rt)* is larger than $\alpha(P_1.rt\text{-}P_2.rt)$. We sort possible crucial differences in descending order of *diff.level* and use C to indicate the sorted set. A difference with the smaller rank in C has a larger probability as the most crucial difference.

4 DEMONSTRATION DETAILS

We plan to present AutoDI in three aspects: 1) we introduce AutoDI design including the three components, 2) we present the whole process to adopt AutoDI to analyze the plan regression, 3) we present how to use AutoDI to analyze one workload. To achieve this, we provide an interactive web interface as a front-end for AutoDI. Through this demo, we adopt JOB benchmark to test TiDB [6], and collect the queries where the default plans are sub-optimal. Note that AutoDI can be general to other database systems with the trivial extensions.

Step 1 - Tool Design Introduction. First, we present the design of AutoDI. This includes the AutoDI's framework, the utility of each component, and how each component works especially, how these components can overcome three challenges mentioned in Sec. 1.



```
ID                                              Est/Act-Rows   EstCost/Runtime
StreamAgg_25                                    1/1           1.68E+8/4.03s
 └─TopN_28                                       1.00/0         1.68E+8/4.03s
   └─IndexHashJoin_42                            2.3E+5/0       1.67E+8/4.03s
     ├─HashJoin_56(Build)                        2.3E+5/0       1.61E+8/4.03s
     │ ├─TableReader_143(Build)[it]             113.00/113      469.57/25.8ms
     │ │ └─TableFullScan_142                     113.00/113      6125.39/-
     │ └─HashJoin_74(Probe)                     2.3E+5/0       1.61E+8/4.03s
     │   ├─TableReader_141(Build)[mi]           1.5E+5/8.8E+3   1.37E+8/4.03s
     │   │ └─Selection_140                     1.5E+5/8.8E+3   2.05E+9/-
     │   │   └─TableFullScan_139               1.4E+7/2.7E+6   2.0E+9/-
     │   └─HashJoin_124(Probe)                 1.3E+6/0       2.30E+7/4.03s
     │     ├─TableReader_135(Build)[ct]       1.00/1         21.59/25.5ms
     │     │ └─Selection_134                   1.00/1          290.00/-
     │     │   └─TableFullScan_133             4.00/4          278.00/-
     │     └─TableReader_138(Probe)[mc]        2.1E+6/2.4E+4   2.23E+7/4.03s
     │       └─Selection_137                    2.1E+6/2.4E+4   2.62E+8/-
     │         └─TableFullScan_136              2.6E+6/2.6E+6   2.54E+8/-
     └─TableReader_36(Probe)[t]                 0.40/0          15.93/0s
       └─Selection_35                            0.40/0          223.57/-
         └─TableRangeScan_34                     1.00/0          0.00/-
```

Plan #1

```
ID                                                   Est/Act-Rows   EstCost/Runtime
HashAgg_24                                           1.00/1         5.45E+8/1.17s
 └─TopN_28                                            1.00/0         5.45E+8/1.17s
   └─IndexJoin_40                                     2.3E+5/0       5.44E+8/1.17s
     ├─IndexJoin_51(Build)                            2.3E+5/0       5.38E+8/1.17s
     │ ├─IndexJoin_64(Build)                          2.3E+5/0       5.33E+8/1.17s
     │ │ ├─IndexHashJoin_104(Build)                   1.3E+6/0       4.56E+7/1.17s
     │ │ │ ├─TableReader_135(Build)[ct]               1.00/1         21.59/8.33us
     │ │ │ │ └─Selection_134                          1.00/1         290.00/-
     │ │ │ │   └─TableFullScan_133                    4.00/4         278.00/-
     │ │ │ └─IndexLookUp_101(Probe)[mc]               1.3E+6/0       4.17E+7/1.17s
     │ │ │   ├─IndexRangeScan_98(Build)               1.6E+6/1.3E+6   0.00/-
     │ │ │   └─Selection_100(Probe)                   1.3E+6/0        0.00/-
     │ │ │     └─TableRowIDScan_99                    1.6E+6/1.3E+6   0.00/-
     │ │ └─IndexLookUp_63(Probe)[mi]                  0.18/0          363.04/0s
     │ │   ├─IndexRangeScan_60(Build)                 17.00/0         0.00/-
     │ │   └─Selection_62(Probe)                      0.18/0          0.00/-
     │ │     └─TableRowIDScan_61                      17.00/0         0.00/-
     │ └─TableReader_48(Probe)[it]                    1.00/0         4.14/0s
     │   └─TableRangeScan_47                          1.00/0         0.00/-
     └─TableReader_36(Probe)[t]                       0.40/0          15.93/0s
       └─Selection_35                                 0.40/0          223.57/-
         └─TableRangeScan_34                          1.00/0          0.00/-
```

Plan #2

Diff ID	Diff Type	Details	Level	Runtime
1	DADiff	TableReader_141, IndexLookUp_63	6	4.03s,0ns
2	ImpDiff	HashJoin_124, IndexHashJoin_104	6	4.03s,1.17s
3	DADiff	TableReader_138, IndexLookUp_101	7	4.03s,1.17s

Diff ID	Reasons
3	Cost model is not accurate at least at this diff. $P1$ ($P2$) has been underestimated (overestimated).
1	With same rank in cost, card. estimation, and runtime, This can happen in $P1$ with (1) user defined hints, (2) different index set
2	Children's card. are different, which may cause different cost in this diff. Cost model is not accurate at least at this mode. $P1$ is underestimated or $P2$ is overestimated.

Diff Type	Count
JODiff	14

Reasons	Count
Cardinality	39

Figure 5: User interface of our demonstration

Step 2 - AutoDI Life Cycle. Next, we go through the life cycle of AutoDI to show how it works. We divide it into three sub-steps: *Step 2.1* Users need to load the new plan regression cases by clicking the 'Load' button in Figure 5. Here, users need to 1) enter the workload ID, 2) assign a unique ID to the new regression case, and 3) load two plans for this plan regression case. With the workload ID, we can conduct an analysis on the all regression cases in it. *Step 2.2* We present how users can use AutoDI to find the differences. Firstly, users need to specify the regression case ID by typing it or clicking the 'View' button, where we will show all the regression cases that have been loaded. After that, the two query plans will be presented in the plan windows. We show the plan in a tree structure, which is easier for user to understand it. The estimated row count, actual row count, estimated cost and the actual runtime are also presented. With such information, users can check whether the latter report on differences and reasons are acceptable. At last, users only need to click the 'Diff' button and AutoDI analyzes the two plans by calling *Preprocessing* and *Find Differences* components. The difference report is displayed in a table as shown in Figure 5, including difference ID (Diff ID), difference type (Diff Type), difference details (Details), difference level, and runtime. In Details column, we show the operator names where one difference is found. In this way, users can quickly locate the difference in the plan tree. *Step 2.3* After differences are obtained in the previous step, users can click the 'INFER' button and AutoDI triggers *Inference* component to find out the reasons behind all differences. The result is displayed in a table and sorted by the importance of the differences according the rule in Sec. 3.2.3. Thus, the result orders of *Find Difference* and *Inference* are different in Figure 5.

Step 3 - Workload Analysis. If one workload has multiple plan regression cases, users can click the 'Workload Analysis' button to dig into it. Users type the workload ID and then click the 'Analyze' button. The whole analysis process in Step 2 is triggered for each plan regression case in the workload. After obtaining all the differences and reasons, summaries on this workload plan regression are displayed. Users can figure out the challenges from these summaries.

ACKNOWLEDGMENTS

This work is supported by a research grant from PingCAP. Zhifeng Bao is supported in part by ARC DP220101434.

REFERENCES

[1] June, 2022. SQL Plan Management. https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-sql-plan-mgmt-19c-5324207.pdf.
[2] June, 2022. SQL Server Compare Showplan. https://docs.microsoft.com/en-us/sql/relational-databases/performance/compare-execution-plans?view=sql-server-ver15.
[3] June, 2022. SQL Tuning Advisor. https://docs.oracle.com/database/121/TGSQL/tgsql_sqltune.htm.
[4] June, 2022. SQL Tuning Advisor. https://docs.microsoft.com/en-us/sql/tools/dta/tutorial-database-engine-tuning-advisor?view=sql-server-ver15.
[5] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2012. Testing the accuracy of query optimizers. In *DBTest*. 11.
[6] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084.
[7] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *VLDB* 9, 3 (2015), 204–215.
[8] Zhan Li, Olga Papaemmanouil, and Mitch Cherniack. 2016. OptMark: A Toolkit for Benchmarking Query Optimizers. In *CIKM*. 2155–2160.
[9] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. In *ICDE*. 1758–1761.
[10] Florian M. Waas, Leo Giakoumakis, and Shin Zhang. 2011. Plan space analysis: an early warning system to detect plan regressions in cost-based optimizers. In *DBTest*. 2.
[11] Mohamed Ziauddin, Dinesh Das, Hong Su, Yali Zhu, and Khaled Yagoub. 2008. Optimizer plan change management: improved stability and performance in Oracle 11g. *PVLDB* 1, 2 (2008), 1346–1355.