Automatic SQL Error Mitigation in Oracle

Krishna Kantikiran Pasupuleti
Oracle America Inc.
Redwood City, CA, USA
kanti.kiran@oracle.com

Jiakun Li
Oracle America Inc.
Redwood City, CA, USA
jiakun.li@oracle.com

Hong Su
Oracle America Inc.
Redwood City, CA, USA
hong.su@oracle.com

Mohamed Ziauddin
Oracle America Inc.
Redwood City, CA, USA
mohamed.ziauddin@oracle.com

ABSTRACT

Despite best coding practices, software bugs are inevitable in a large codebase. In traditional databases, when errors occur during query processing, they disrupt user workflow until workarounds are found and applied. Manual identification of workarounds often relies on a trial-and-error method. The process is not only time-consuming but also requires domain expertise that users are often lacking. In this paper, we propose a framework to automatically mitigate errors that occur during query compilation (including optimization and code generation) without any user intervention. An error is intercepted by the database internally, a workaround is identified for it, and the query is recompiled using the workaround. The entire process remains transparent to the user with the query being executed seamlessly. The proposed technique handles SQL errors during query compilation and provides three types of mitigation strategies – i) quickly failover to one of the readily-available historical plans for the statement ii) apply targeted error-correcting directives (hints) identified from the optimizer context at the time of the error iii) modify the global configuration of the optimizer using hints.

This feature has been implemented and will be released in an upcoming version of Oracle Autonomous Database.

PVLDB Reference Format:
Krishna Kantikiran Pasupuleti, Jiakun Li, Hong Su, and Mohamed Ziauddin. Automatic SQL Error Mitigation in Oracle. PVLDB, 16(12): 3835 - 3847, 2023.
doi:10.14778/3611540.3611568

1 INTRODUCTION

One of the most appealing aspects of cloud offerings of database management systems is savings in cost. A cloud offering helps customers reduce not only their investment in proprietary hardware, but also the labor cost involved in database administration. A fully managed database service is the industrial trend nowadays and offers a work out of box" experience to customers. It minimizes human intervention by automating database provisioning, software upgrade, security, performance tuning etc. In addition to the labor cost savings, such services further aim to eliminate human errors, offer insider's expertise that a database administrator (DBA) often lacks and optimize user experience.

SQL query failures that cause interruptions in the workflow due to errors in software undoubtedly lead to poor user experience. However, such errors are inevitable. Database systems are complex pieces of software that are built from large codebases. With constant innovation, as more features are developed and modularized, their interaction can sometimes lead to unexpected errors. When an error happens in a traditional on-premises database environment, DBAs must quickly find a workaround to bypass the faulty code path. Without access to the source code, they often use a trial-and-error method to figure out a workaround. Additionally, if multiple workarounds exist, they need to pick the one that gives the optimal performance. Such an iterative process is time-consuming, and its effectiveness relies heavily on the expertise of the DBAs. Moreover, after a workaround has been identified, the customer must re-run the query that has previously failed to resume the workflow. Finally, to get the root cause fixed, the customer needs to contact the vendor's support team, provide the details of the error and wait for a fix. The amount of human intervention involved in the whole process leads to high turnaround time and support fee.

1.1 Automatic Error Mitigation Framework

Using auto error mitigation (AEM) framework, we address the above pain points in the context of Oracle's fully managed database service on cloud called Autonomous Database (ADB). The proposed design intercepts errors and attempts to mitigate them by automatically discovering and applying workarounds, all transparent to the user. Meanwhile the system maintains information about the error and its workaround, that are immediately provided to Oracle support for better observability and diagnosis, thus contributing to quicker root-cause fixes. In the first phase of this endeavor, we focus on SQL query failures due to assertions. Assertions are commonly used in programming languages. An assertion failure indicates certain pre-conditions have not been met and program execution must be stopped before more harm is done. For example, an assertion can be added to check whether a pointer is null before its de-reference and is fired if the pointer is found to be null. In our experience, assertions account for a large percent of bugs filed for SQL query compilation failures. Therefore, our initial focus is on mitigating errors during query compilation that spans query optimization and

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611568

code generation. Error mitigation during query execution has its distinct challenges, such as cleaning up processes and restarting them (parallel execution further complicates the situation), dealing with partial results that have been generated, etc. However, our framework is designed with extensibility in mind so that it can accommodate failures during query execution in the future.

The following are the main design goals for the first version of the feature.

End-user transparency: When an error occurs, we suppress it internally and automatically find a workaround. If a workaround exists, the query continues to compile and execute. The user may experience a delay in query compilation but doesn't notice the error.

Reasonable query compilation time: We impose an upper bound on the time spent in error mitigation. This is because a query that takes a very long time to compile may itself create a poor user experience even if error mitigation is successful.

No repetitive mitigation: It is important to avoid repeated attempts to mitigate errors for a query when prior attempts have been unsuccessful. We need a fair balance between avoiding repeated attempts and ensuring timely error mitigation.

Integration with Oracle's diagnostic framework: A workaround is not the eventual solution as it may not always be as performant. Eventually a code fix is required so that the workaround can be removed. Therefore, the internal diagnostic information pertaining to the original assertion failure (called "incident" in Oracle) is generated regardless of whether error mitigation is successful or not. Additional information regarding how the workaround was identified (e.g., the internal strategies that have been tried and whether they were successful) is logged so that it would i) help developers understand, triage and debug the problem better, ii) expedite identification of duplicate bugs etc. Such information can also be mined by DevOps for various purposes like monitoring the quality of new features, identifying problems and proactively disabling a feature or withdrawing a regressed patch to prevent problems from affecting more customers.

In summary, the design is based on the premise that a query that uses a workaround and continues to seamlessly execute in a reasonable time creates a superior user experience than a query that fails and possibly entails human intervention.

While there is literature on automatic error corrections in systems software and hardware, to the best of our knowledge our work is the first that attempts automatic error mitigation in SQL query processing in a database system. We present our survey of Related Work in Section 7.

We have implemented the automatic error mitigation feature in Oracle autonomous database. Our testing and customer support teams have verified its efficacy on internal and customer reported bugs. Although some of our ideas rely on the expressive power of hint framework in Oracle, the technique is general and can be extended to other database systems using hints and other knobs that may be available in them.

2 OVERVIEW

In this section we briefly describe certain concepts that are the building blocks of our solution before presenting an overview of the proposed auto error mitigation (AEM) framework. Section 3 and 4 describe in detail how these components work together to achieve error transparency.

2.1 SQL Hints

SQL Hints are directives that obligate the optimizer to honor actions they specify and can be used to make corrections when the optimizer fails to produce a desirable plan. In Oracle, hints have the semantic flexibility to dictate an action at various granular levels of a SQL statement, e.g., a designated table, a specific query block or the entire statement. All operations that constitute a query plan can be represented and controlled by corresponding hints. We propose using this versatility and precision of hints to disable the specific operation that is active at the time of error.

2.2 Optimizer Version Control

Oracle database provides a mechanism to control the version of the optimizer that can be used to compile queries. The version can be specified using a hint or a parameter. All new features, enhancements and bug fixes made to the optimizer source code are tied to their respective optimizer versions. When a query is hinted with a version, the optimizer compiles the query using the hinted version's capability.

2.3 Automatic Error Mitigation (AEM)

We propose two mechanisms to mitigate query compilation errors: one that uses historical alternate plans for query failover and the other that identifies and uses error-correcting hints.

Figure 1 shows the overview of the control flow of AEM. The left-most column depicts the user session that submits a query to the database and awaits results. After necessary pre-processing like syntactic parsing, the query compiler (middle column) takes over and compiles the query, performing a series of tasks including optimization and code generation. If an error is encountered during this process, AEM intercepts the error and attempts error mitigation via a sequence of actions, as shown in the middle column. At certain points, AEM reads from and writes to the storage layer (right-most column) to realize functionalities such as plan retrieval, workaround generation, operational monitoring and diagnostics etc.

A new exception handler is instantiated to catch errors raised during compilation. When an error is raised within the protected section, the following sequence of events take place.

1. At the time of the error, fine-grained candidate hints are generated based on the context and stored in dictionary tables.
2. Subsequently, an attempt is made to retrieve alternate plans available in the system. If any of the alternate plan compiles successfully, the statement proceeds to execution.
3. If no alternate plans exist or none of them succeeds in compilation, the AEM driver applies the fine-grained candidate hints generated in step 1. If the hints fail to mitigate the error, the driver uses hints to modify certain global configurations (e.g., reverting the optimizer to an older version) and recompiles the statement.
4. If any workaround hint succeeds, the statement is compiled with the hint and proceeds to execution. An object

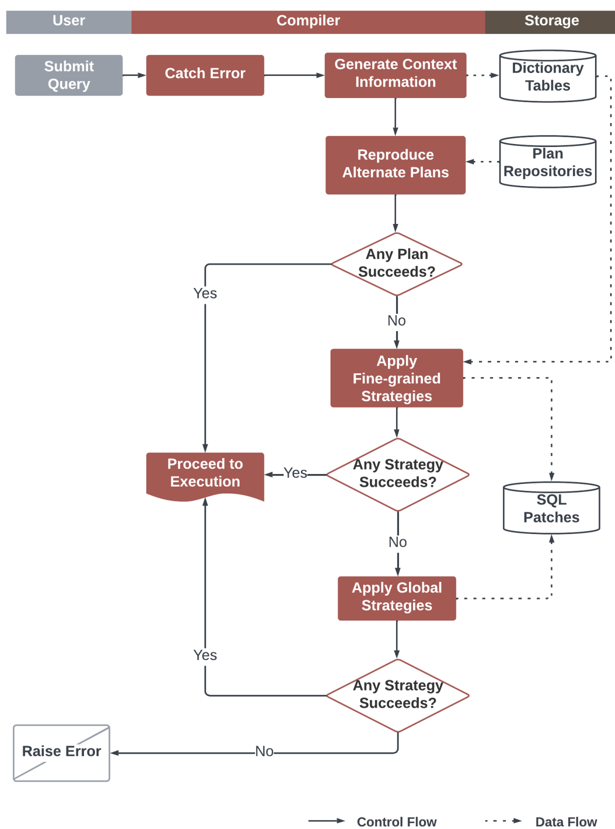

Figure 1: Overview of Automatic Error Mitigation (AEM).

containing the hint (called a SQL Patch) is generated and persisted so that it is used for future compilations of the same statement as described in Section 4.

(5) If no workaround is found, the error is returned to the user.

Handling Recursive Errors

It is possible that the process of error mitigation itself may encounter errors. Such errors can be classified as "recursive" as they are not related to the original (top) error that is being mitigated. The framework's exception handler intercepts these recursive errors and ignores them deeming the underlying operation to have failed. For instance, if an error occurs while choosing an alternate plan, the process moves to the next available alternate plan.

In subsequent sections we describe the two core parts of our design in detail.

3 AEM USING ALTERNATE PLAN

When a SQL error occurs, the optimizer first attempts to use an alternate plan for the SQL statement. This is a quick way to avoid the error when an alternate plan that compiled successfully exists. Oracle database captures and stores all historical plans generated for each statement in various sources e.g., Auto SQL Tuning Set (Auto STS) [6], Automatic Workload Repository (AWR) [14] and in-memory plan cache. These sources together form a pool of SQL plan repositories from which AEM retrieves alternate plans for a SQL statement. Each plan is logically represented by a collection of hints called the plan outline. A plan outline specifies the sequence of hints (aka compilation steps) that when applied during compilation of a statement, generates (or reproduces) the same plan. [25] describes how plans can be reproduced deterministically using plan outlines.

3.1 Switch to Alternate Plan

In response to an error, the optimizer retrieves alternate plans from plan repository and attempts to reproduce each of them using their corresponding plan outlines. The plans are examined in temporal order from the newest to oldest, as newer plans are generated using fresher statistics and are likely to be better. The first plan that compiles successfully is chosen and used ensuring a quick switch to the alternate plan. However, the optimizer can explore more alternate plans if available, cost them and choose the cheapest among them ensuring better plan quality at the expense of additional latency to the user query. We defer the exploration of this aspect to future work.

3.2 Benefit

Plan outline driven compilation is quick as the outline contains a hint for every step of compilation obligating the optimizer to honor the hint. As a result, the optimizer doesn't explore the potentially vast search space of optimization and quickly generates the intended plan.

The other benefit of relying on alternate plans is assurance of the quality of the plan. As alternate plans correspond to the best plans generated by the optimizer in the past, their performance tends to be reasonably good and predictable.

4 AEM USING SQL PATCH

When alternate plans are not available for a SQL statement, a workaround needs to be identified in order to overcome the error and compile the statement. A successful workaround depends on the nature of the error and the context in which it is raised. For example, some errors may be specific to the logical transformation [2] that is being attempted on the query at the time of the error whereas others may disappear when certain set of features are disabled for the entire query.

This section presents techniques to identify effective workarounds. They contain hints identified at different levels of granularity that help in avoiding error conditions. After a workaround is identified, an object called "SQL Patch" is created for the statement that contains the corresponding set of hint(s) that implement the workaround. When the statement is recompiled, the SQL Patch is attached to it and the hints present in the patch steer the optimizer away from the error generating conditions.

The SQL Patch generated for the statement is persisted in the database so that all future compilations benefit from it. As this method is a one-time mitigation effort, the amortized (over a period) cost of time spent in identifying a workaround is low.

```
Q1:
SELECT SUM(s.unique1)              (SEL$1)
FROM T_10K s, T_5K d1,
     (SELECT *                     (SEL$2)
      FROM T_5K d2
      WHERE unique2 = 3) V1
WHERE s.ten = d1.ten
  AND s.thousand = V1.thousand
  AND d1.hundred = V1.hundred
GROUP BY d1.ten;
```

Figure 2: Query Q1 with two query blocks and their initial names.

```
-----------------------------------------------------------
| Id  | Operation                         | Name           |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                  |                |
|   1 |  SORT GROUP BY NOSORT             |                |
|   2 |   MERGE JOIN                      |                |
|   3 |    SORT JOIN                      |                |
|   4 |     VIEW                          | VW_GBF_2       |
|   5 |      HASH GROUP BY                |                |
|   6 |       INDEX FAST FULL SCAN        | T_5K_CONCAT3   |
|   7 |    SORT JOIN                      |                |
|   8 |     VIEW                          | VW_GBC_1       |
|   9 |      HASH GROUP BY                |                |
|  10 |       NESTED LOOPS                |                |
|  11 |        TABLE ACCESS BY INDEX ROWID| T_5K           |
|  12 |     INDEX UNIQUE SCAN             | T_5K_UNIQUE2   |
|  13 |        TABLE ACCESS BY INDEX ROWID| T_10K          |
|  14 |     INDEX RANGE SCAN              | T_10K_THOUSAND |
-----------------------------------------------------------
```

Figure 3: Final Plan of Q1 with two grouped views.

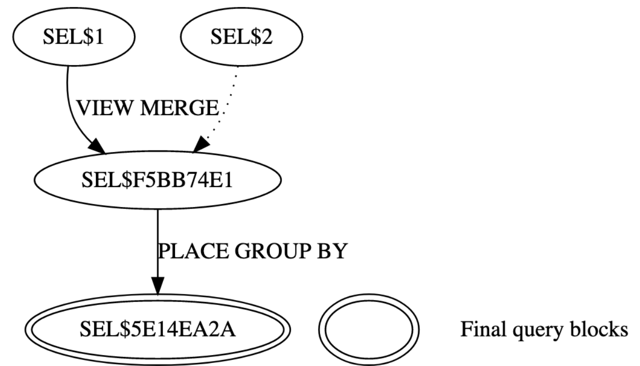

Figure 4: Query Block Registry graph for Q1.

This section is organized as follows. Section 4.1 gives an overview of transformations and how query blocks are "named" during compilation. Section 4.2 describes how fine-grained candidate hints can be targeted using the query block names. Section 4.3 gives a qualitative assessment of how fine-grained hints can affect query plans. Section 4.4 describes the process of identification of a successful workaround using the candidate hints and Section 4.5 addresses the issue of time budget for error mitigation.

4.1 Transformations

Transformations are logical query rewrite techniques that the optimizer uses to transform a query to its semantically equivalent but more efficient form. A transformation is finalized using a cost-based framework where the optimizer explores and costs different ways (called "states") in which it can apply the transformation on the query and chooses the cheapest among them (called the best state). The complete set of states corresponds to the total state space of the transformation. The optimizer may explore the entire state space or a sub-space depending on the search algorithm chosen based on budget. We show a sample transformation sequence and describe how query block names represent the essence of this sequence.

Sample Sequence of Transformations

During optimization, multiple transformations may be applied on a query one after the other, leading to a final optimal representation. Consider a query Q1 that has two query blocks SEL$1 and SEL$2 as shown in Figure 2. The optimizer initially "merges" the view V1 (SEL$2) into its outer query block SEL$1. After this transformation, SEL$1 has three tables s, d1 and d2. Subsequently, the joins between these tables are optimized by grouping their rows prior to performing the joins. Grouping may significantly reduce the number of rows that participate in a join making it more efficient. However, the cost of grouping must also be considered before finalizing this option. This cost-based transformation is called "Group-By Placement" (GBP). In Q1, GBP transformation is applied on the outer query block generating the final plan shown in Figure 3. The plan contains two group-by views, VW_GBC_1 and VW_GBF_2, over different sets of tables.

The sequence of transformations for this query is i) View Merge followed by ii) Group-By Placement (GBP).

Query Block Names and Registry

Query blocks are assigned names at the beginning of parse. The names are prefixed with string "SEL$" and suffixed with a monotonically increasing sequence number. Each transformation applied on a query block generates a new name for it using the result of a deterministic hash function. For instance, when a transformation X is applied on a query block, a new name is assigned that is based on a hash signature computed using i) the identity of the transformation X ii) parameters of X (for example the tables or other objects involved) and iii) the old name of the query block. At the end of compilation, the names of the query blocks are their final names. As each hash signature depends on its prior signature, a signature at any point indirectly captures the entire sequence of transformations that have been applied on the query block until that point. The names generated for all query blocks in a SQL statement are tracked internally in a directed acyclic graph called Query Block Registry whose vertices are names and edges are transformations. The graph for Q1 is shown in Figure 4 (edges representing some internal operations are not shown for conceptual clarity). In the beginning, the query blocks are SEL$1 and SEL$2. At the end of transformations, they are replaced by the final query block, SEL$5E14EA2A. (The dotted edge represents additional argument to the View Merge transformation).

4.2 Candidate Hint Generation

A candidate hint is a fine-grained hint that is generated based on the premise that an error is correlated to the context that exists at that time. It is a "negative" hint that disables a transformation or its state(s) that is active at the time of the error. The first argument of the hint is a query block name indicating that the hint can only be applied to the query block with the specified name. For example, the hint NO_UNNEST(@SEL$2) disables the transformation named "UNNEST" on a query block whose name is SEL$2.

As described earlier, the name of a query block is a proxy to all prior transformations that happened on and consequently, it is a proxy to the structure of the query block. Errors are often correlated to query block structure. This is because operations during compilation modify query blocks and their associated structures in certain ways and code assertions enforce the structural sanity. Hence, a negative hint that uses the name of the query block precisely targets the error. In subsequent compilations, if the query block gets a different name its structure will be different, and the hint will not be applied which is desirable.

Fine-grained hints are less disruptive to the quality of query plans and have the following advantages.

- They disable specific states of a transformation keeping the rest of the state space open to the optimizer for exploration and costing.
- They are applied at a relatively finer granularity of query blocks without affecting the entire SQL statement.
- They precisely capture the structural conditions under which an error must be mitigated using query block names as proxies.

When an error occurs during compilation, as a first step, an "incident" is generated and information about the error is logged in system tables under the incident. Incidents have unique identifiers that can be used by the system to retrieve all the necessary information about them. After this step, control reaches the exception handler for automatic error mitigation. The handler checks if the error qualifies for mitigation based on the type of the statement and parameter settings. Subsequently, it invokes the candidate generation code that uses the context of the error to generate hints that may be successful in mitigating the error.

There are three types of fine-grained hints – i) hints that disable specific states of a transformation ii) hints that disable all states of a transformation and iii) hints that disable a sub-space of a transformation.

4.2.1 Disable a Specific State of a Transformation. In this section, we explain how fine-grained hints are generated for specific states of a transformation using an example that shows the state space of GBP. Consider query Q1 (from Section 4.1) whose query block SEL$1 is first transformed by merging the view query block SEL$2 into it. The equivalent query representation Q1M, after the transformation is shown in Figure 5(along with the query block's name at that point). Subsequently, GBP transformation is applied on the merged query block.

```
Q1M:
SELECT SUM(s.unique1)                  (SEL$F5BB74E1)
FROM T_10K s, T_5K d1, T_5K d2
WHERE s.ten = d1.ten
  AND s.thousand = d2.thousand
  AND d1.hundred = d2.hundred
  AND d2.unique2 = 3
GROUP BY d1.ten;
```

Figure 5: Representation of Q1 after view-merging.

State Space of GBP Transformation

GBP divides the tables in a query block (tables *s*, *d1* and *d2* in Q1M) into one or two sets and creates Group-by views over each set.

- One set is called a "coalesced" group (C) and must necessarily contain all the tables appearing in the aggregate functions of the query block (table s as it appears in SUM()) along with any combination of other tables (tables *d1*, *d2*).
- The other set is called a "factored" group (F) that contains any combination of tables that do not appear in the aggregates of the query block (tables *d1*, *d2*).

Coalesced groups create partial aggregates while factored groups create multiplicative factors. Either C or F may be empty and hence not present in the final plan. In Q1M, tables *s*, *d1* and *d2* can be divided into sets C and F in various ways adhering to the above conditions thus generating a combinatorial state space. The state space explored by the optimizer is shown in Table 1. The columns "C" and "F" correspond to coalesced and factored groups and indicate the tables that will be present in them. Table *s* appears in the aggregate function SUM() and hence always appears in set C.

Table 1: State Space of GBP Transformation for Query Q1M.

Tables {*s d1 d2*}	Coalesced Group C	Factored Group F
{1 1 1}	{*s, d1, d2*}	Ø
{1 1 0}	{*s, d1*}	{*d2*}
{1 0 1}	{*s, d2*}	{*d1*}
{1 0 0}	{*s*}	{*d1,d2*}
{0 * *}	Invalid states (Table *s* appears in SUM(); must be in C)	

Query Q1T in Figure 6 shows the transformed query corresponding to state {1 0 1} (view VW_GBC_1 corresponds to set C and view VW_GBF_2 corresponds to set F of the state).

If an error occurs while the optimizer is processing state {1 0 0} shown in red in Table 1, AEM generates a candidate fine-grained hint, **"NO_PLACE_GROUP_BY (@SEL$F5BB74E1 (s) (d1 d2))"** to disable this state. (The query block name for Q1M is SEL$F5BB74E1 as shown in Figure 4 and Figure 5. The name of the negative hint for GBP is NO_PLACE_GROUP_BY).

```
Q1T:
SELECT SUM(VW_GBC_1.c3 * VW_GBF_2.c3)
FROM
   (SELECT d1.hundred c1, d1.ten c2, COUNT(*) c3
    FROM T_5K d1
    GROUP BY d1.hundred, d1.ten) VW_GBF_2,
   (SELECT d2.hundred c1, s.ten c2, SUM(s.unique1) c3
    FROM T_10K s, T_5K d2
    WHERE s.thousand = d2.thousand
      AND d2.unique2 = 3
    GROUP BY d2.hundred, s.ten) VW_GBC_1
WHERE VW_GBC_1.c1 = VW_GBF_2.c1
  AND VW_GBC_1.c2 = VW_GBF_2.c2
GROUP BY VW_GBF_2.c2;
```

Figure 6: Query Q1M transformed according to state {1 0 1}.

```
Q2:
SELECT t1.ten                                          (SEL$1)
FROM T_10K t1, T_5K t2
WHERE t1.ten = t2.ten
  AND EXISTS(SELECT 1 FROM T_4K t3                     (SEL$2)
             WHERE t3.thousand = t2.thousand)
  AND NOT EXISTS(SELECT 1 FROM T1_100 t4               (SEL$3)
                 WHERE t4.ten = t1.ten)
  AND EXISTS (SELECT 1 FROM G_4K t5                    (SEL$4)
              WHERE t5.hundred = t2.hundred);
```

Figure 7: Query Q2 with three sub-queries.

4.2.2 Disable All States of a Transformation. A hint that disables all states of a transformation is another type of fine-grained hint that is generated and is relatively more disruptive. Taking the earlier example of GBP transformation, the negative hint **NO_PLACE_GROUP_BY(@SEL$F5BB74E1)** disables all states of GBP in the query block SEL$F5BB74E1 and hence the transformation cannot take place in it.

4.2.3 Disable Sub-space of a Transformation. For certain transformations it is possible to generate a negative hint that disables a subset of the entire transformation state space (or a sub-space). For example, consider a query Q2 shown in Figure 7 that contains three subqueries SEL$2, SEL$3 and SEL$4 in an outer query block SEL$1. "Subquery unnesting" (SU) is a cost-based transformation that decorrelates one or more subqueries converting them into join(s) in the outer query block. The state space of SU is defined over the three subqueries SEL$2, SEL$3 and SEL$4. An equivalent binary form (like Table 1) consists of all combinations of $\{x_1, x_2, x_3\}$, where each $x_i \in \{0, 1\}$ and denotes whether the corresponding subquery is unnested or not. Hence, there is a total of 8 states.

If an error occurs in state {1, 0, 1}, two fine-grained hints - **NO_UNNEST(@SEL$2)** and **NO_UNNEST(@SEL$4)** - are generated one for each subquery. Each of these hints disables one half of the state space. For instance, NO_UNNEST(@SEL$2) disables unnesting of subquery SEL$2, i.e., disables all states $\{1, x_2, x_3\}$ where $\{x_2, x_3\} \in \{0, 1\}$. Similarly, NO_UNNEST(@SEL$4) disables all states $\{x_1, x_2, 1\}$ where $\{x_1, x_2\} \in \{0, 1\}$.

4.3 Impact of Fine-Grained Hints on Query Plan

In this section we analyze the circumstances under which a query plan generated with a fine-grained hint, referred to as "mitigated plan", may differ from the plan generated when there is no error (e.g., after the error is fixed), referred to as "no-error plan". This helps us determine circumstances under which these plans may be different and consequently have different performance. The analysis is presented for a mitigation strategy that uses a fine-grained hint that disables one state of a transformation; however, other types of fine-grained hints also follow a similar pattern with the caveat that they can be more disruptive.

Consider a sample transformation T. There are two possibilities for the no-error plan - either the plan contains T or it doesn't contain T. Further, consider that an error occurs in state s1 of a transformation T and a fine-grained hint that disables the state s1 is generated.

If the no-error plan contains T, the mitigated plan may sometimes differ from the no-error plan as follows. If s1 is any state other than the best state (a transformation with N states, has N-1 non-best states and 1 best state), the mitigated plan is the same as the no-error plan because disabling a non-best state doesn't prevent the optimizer from choosing the best state. In the less likely situation where s1 is the best state (that is disabled), the optimizer chooses the second-best state for T. It is possible that this choice of state results in a different sequence of downstream transformations that take place after T. The quality of the mitigated plan thus depends on the extent of the difference between the best and the second-best states of T as well as the cumulative effect of the downstream decisions that follow transformation T.

If the no-error plan does not contain T because it is expensive, the mitigated plan also doesn't choose T irrespective of the state that is disabled because all of them would be expensive. The combinations are shown in Table 2.

Table 2: State Space of GBP Transformation for Query Q1M.

Mitigated Plan same No-Error Plan?	Error occurs in Non-best States of Transformation T	Error occurs in Best State of Transformation T
Transformation T is NOT chosen in No-Error Plan	✓	✓
Transformation T is chosen in No-Error Plan	✓	× (Plan uses second-best state and possibly different transformations downstream)

4.3.1 Non-transformation errors. Fine-grained hints can also be generated for errors that happen post-transformations during physical optimization. For instance, an assertion failure may happen

while generating a hash join on a particular table and can possibly be fixed by avoiding hash join and using a different join method. Or it may go away if the build and the probe tables of the hash join are switched. Another possibility is the error may depend on a specific join order and may not occur when the same hash join is attempted in a different join order. There can be many such candidates for fine-grained hints. We plan to extend our ideas in the future to handle such errors.

4.4 AEM Driver

After candidate hints for mitigation are generated, the AEM driver component needs to identify from the candidates a hint that successfully compiles the query. The driver performs the following.

- Identifies a workaround: It uses two strategies in the following order i) fine-grained candidate hints generated based on context ii) optimizer version umbrella hints to compile the query with different optimizer versions.
- Creates a SQL Patch for the workaround: It creates and persists a SQL Patch object for the statement that contains the error-mitigating hint (fine-grained hint or optimizer version hint).

4.4.1 Identify a Workaround. The driver first retrieves the candidate fine-grained hints for the SQL statement and allocates a global time budget for identification of a workaround (Section 4.5). It prioritizes the fine-grained hints and considers them in a specific order - from the least disruptive hint to the most disruptive hint and iteratively compiles the SQL statement using one hint at a time. A hint that disables a single state of a transformation is less disruptive than a hint that disables a sub-space of the transformation and so on. Each iteration is also assigned a dynamic time limit computed based on the cumulative time already consumed and the remaining global time budget. If the error persists after all the fine-grained hints are attempted, the driver uses optimizer version hints beginning from the latest version and proceeding to older versions. This is because an optimizer version hint that specifies a version number modifies the capability of the optimizer to match that of the identified version; so, the older the version number, the less capable the optimizer may relatively be, of generating good query plans. After a workaround is identified the AEM driver creates a SQL Patch for the query in persistent store of the database.

Like in the case of alternate plan exploration (Section 3.1), the driver stops compiling the query as soon as it identifies a working hint. Hence, the order in which the hints are considered is important. Alternatively, the driver can proceed further and consider all working hints and choose the one that generates the best(cheapest) plan. This comes at the expense of query latency; we defer the details of this exploration to our future work.

The AEM driver process is interruptible and can perform the above activity incrementally across multiple iterations. It maintains the required state information that allows it to determine its prior progress and restart from that point. It also logs information about the number of iterations, the hints attempted during these iterations and their outcome, etc. in the system tables in XML format. DBA views are provided that allow users to query these tables and analyze the information.

Optimizer version hints are not based on the context in which an error occurred; rather they are control knobs that affect the entire statement. Besides optimizer version, there are other context-free hints that can be used (e.g., parallelism of the query can be altered using a hint). We plan to explore their usage in future.

4.5 Time Limit and Repetitive Mitigation

If the error mitigation process takes a long time, it leads to an undesirable user experience and must be avoided. AEM driver uses an automatically derived configurable time limit and aborts the process once the allotted time expires. In our experience this helps deal with extreme cases that present long compilation times but doesn't impact most of the real-world queries. If the query has been executed before, the time limit is derived by considering its historical compilation and execution statistics. The average compilation time indicates the expected time the query takes to compile that needs to be considered when finalizing a budget. The average execution time is used to identify the overhead of compilation in the total elapsed time for the query. For a long running query, it is reasonable to allocate more budget in the expectation that the increase in compilation time has minimal effect on the total elapsed time. Also, as queries are compiled once and executed many times, the additional compilation overhead is amortized.

Similarly, it is not useful to attempt to mitigate errors in quick succession for the same SQL statement. If the driver was unsuccessful in identifying a workaround for a SQL, it is less likely that a second attempt in a short span of time will lead to a different outcome, and resources are better spent elsewhere in the database. The system tables track information (e.g., timestamp) about prior mitigations that were attempted for each SQL statement, and the system lets a specific interval of time to pass before re-attempting mitigation on the same statement.

5 OBSERVABILITY

AEM telemetry resides in multiple locations, including query execution plans, trace files and system tables. Logging AEM telemetry serves both informative and functional purposes. They provide user transparency, facilitate troubleshooting, and can be mined for a global solution to a prevalent issue.

Query Execution Plan Display: Oracle stores query execution plans in shared memory accessible from all processes. Users with the right privileges can use an API to display the plan information. When a user displays the plan of a query that was mitigated by AEM, the plan shows either the name of the alternate plan or the name of the SQL patch depending on the type of mitigation.

Dictionary Tables and Views: The AEM module logs detailed information about error mitigation actions and persists the information in system tables and trace files irrespective of whether mitigation succeeds or not. DBA views are created over these tables to provide user-friendly access to the relevant information like the erring SQL, whether AEM was attempted, each strategy attempted, and time spent in it as well as the total time etc.

Trace Files: AEM module logs information about errors into trace files. The trace files also containing other diagnostics generated for an incident and are automatically shipped to Oracle development teams for further investigation.

Database users, Cloud Operations and database developers benefit from the detailed telemetry. Database users can query the DBA views to get more information about mitigation results. Cloud Operations teams can also mine this information potentially across multiple tenants to monitor the health of SQL statements and take corrective actions if necessary. For example, if it is evident that a particular transformation causes a lot of SQL statements to fail, it can be disabled for the database. AEM metadata information serves as valuable feedback to Oracle even when AEM fails to find a workaround. For example, it helps rule out that a newly installed fix is the culprit if AEM has proven that an old optimizer version bypassing that fix still does not solve the problem. If AEM times out, the information about search strategies attempted and the time taken helps development teams design better mitigation strategies.

Functional Use of Metadata: AEM also relies on the metadata logged in system tables for some of its functionality. For example, it uses the timestamp information on the last mitigation attempt for a SQL statement to decide if another mitigation can proceed.

6 EXPERIMENTAL ANALYSIS

6.1 In-house Testing Approach

To test the functionality of AEM, we design an artificial error-raising mechanism that simulates real-world failures and covers a wide scope of code paths of the optimizer. During testing, artificial errors are raised in a controlled manner at designated locations, so that it is known a priori whether a workaround exists and what the workaround ought to be.

Three Dimensions of Search Space

To achieve broad coverage of query compilation code in tests, it is important to design external controls for error locations along the same dimensions as the search space of the query optimizer. The entire set of reachable error locations through these controls should form a space similar in shape and size to the search space of the optimizer.

The main dimensions to the search space of an Oracle optimizer are:

- Transformations: The optimizer searches through a series of cost-based and heuristic-based transformations. E.g., Subquery Unnesting, Group-by Placement, Star Transformation etc.
- Query Blocks: The optimizer applies transformations to all eligible query blocks.
- States of a Transformation: For each combination of a query block and a transformation, the optimizer visits a set of states that denote combinations of relevant objects in the query block and picks the best (cheapest) state.

We introduce new control parameters for errors that follow these dimensions.

- A parameter whose value is set to a specific transformation forces an assertion within that transformation code to fail.
- A parameter whose value is set to a specific query block forces assertion failure for that query block.
- A third parameter that can be set to a state number forces an error within that state.

To further randomize the code locations within a state of a transformation where error needs to be raised, we use a fourth parameter that specifies a number between 0 and N (a fixed number). As code executes, it applies a hash function on the error message of every assertion generating a number between 0 and N and fires the assertion if it matches the parameter input.

Different combinations of the above parameters allow us to precisely control the locations where errors can be force-raised for testing AEM.

Testing AEM using SQL Patches

Using the parameters described, it is possible to raise a vast range of errors including i) errors that can be mitigated by fine-grained hints, ii) errors that cannot be mitigated by fine-grained hints but can be mitigated by global controls like optimizer version and iii) errors that cannot be mitigated at all.

For example, if the parameters are set to force an error in a specific state of a transformation, there exists a context-aware strategy that skips the state and avoids the error. If they are set to force an error in a transformation irrespective of state and query block, the circumvention uses an optimizer version that predates the erring transformation. If the parameters are set to raise errors at a point that is reached even by the oldest version of the optimizer, no workaround exists.

Testing AEM using Alternate Plans

During exploration of multiple alternate plans, the parameters are used to filter a subset of alternate plans. For example, if a subset of alternate plans contains Group-by Placement transformation, we set the parameters to raise error during Group-by Placement so that those alternate plans cannot be used.

We also have a parameter that controls the type of mitigation that is attempted – alternate plan only, SQL Patch only, or both.

6.2 Benefit of Fine-Grained Hints

In this section we demonstrate how using context-based fine-grained hints can lead to better query plans when compared to context-free hints like optimizer version. For illustrative purpose, we conducted an experiment using TPCDS query 4 (Q4) on 1GB testbed on Intel X86 machine. In the experiment, we forced an error in multiple states of GBP transformation and observed the effect on the query plans. We GBP as a representative transaction to discuss as its concepts have been described earlier and it applies to Q4 (among others). Q4 has two branches of a union-all set query block, both of which are potential candidates for GBP. The text of the query is not shown for lack of space; readers can refer to the TPCDS specification. The benefit of using fine-grained hints for errors can be seen in Figure 8 that shows how the cost of the final plan varies according to the hint used.

The unit of cost is an abstract quantity derived from a combination of CPU cycles, IO operations and memory overhead. The original query plan (when there is no error) has a cost of 1872K. If an error occurs when the best state of GBP transformation is being explored in query block SEL$1, the state must be disabled. This increases the cost slightly to 1875K when the transformation

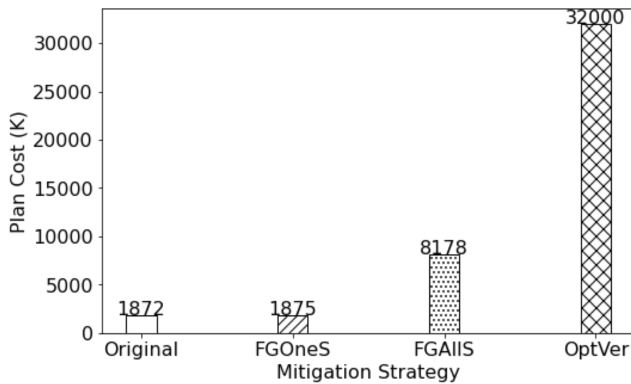

Figure 8: TPCDS Q4: Comparison of the costs of the original plan ("Original") and three mitigated plans. Three mitigated plans are generated by a fine-grained strategy that disables the erring state ("FGOneS"[1]), a fine-grained strategy that disables all states ("FGAllS"[2]) and a global strategy that reverts to an older version of the optimizer ("OptVer"[3]).

chooses the second-best state. If, however, we disable the transformation completely on SEL$1, the cost would be 8178K. Finally, if an older optimizer version is used as the mitigation strategy, the cost increases further to 32M.

We show the relevant portion of the query plans that are generated using each mitigation strategy and discuss their impact. The original plan of Q4 is shown in Figure 9 (some plan lines that are not related to the GBP transformation are omitted for brevity). The query blocks SEL$1 and SEL$2 are annotated in blue. Both have GBP with their respective best states as shown in the plan. In SEL$1, a grouped view (VW_GBC_11) is created over the result of the join between tables DATE_DIM and STORE_SALES and this view is joined with the table CUSTOMER. Similarly, in SEL$2 a grouped view (VW_GBC_22) is created over the result of the join between tables DATE_DIM and CATALOG_SALES and this view is joined with the table CUSTOMER.

Let's assume that an error occurs when optimizer attempts GBP transformation in query block SEL$1.

Strategy-1: Fine-Grained Hint - Disable best state

As shown in Table 2, the worst-case situation is an error occurs during exploration of the best state in a query block and the state must be disabled. In SEL$1 of Q4, the negative hint generated to disable the best state (of the plan in Figure 10) is NO_PLACE_GROUP _BY(@SEL$1 (STORE_SALES DATE_DIM)). The new plan is shown in Figure 11. It can be observed that there is no impact on the GBP transformation in SEL$2. However, in SEL$1, GBP was chosen using the second-best state in which two grouped views (VW_GBF_12 and VW_GBC_11) are created - one on table CUSTOMER and the other on the result of the join between tables DATE_DIM and STORE_SALES. Finally, both these views are joined. This increases the cost of the entire plan to 1875K (top line in the plan).

Id	Operation	Name	Cost (%CPU)
0	SELECT STATEMENT		**1872K**(11)
5	UNION-ALL		
10	MERGE JOIN	(SEL$1)	1074K (12)
15	VIEW	VW_GBC_11	888K (13)
16	HASH GROUP BY		888K (13)
20	HASH JOIN		88030 (57)
25	TABLE ACCESS FULL	DATE_DIM	95 (33)
27	TABLE ACCESS FULL	STORE_SALES	80280 (52)
28	SORT JOIN		180K (5)
32	TABLE ACCESS FULL	CUSTOMER	3589 (31)
37	MERGE JOIN	(SEL$2)	693K (11)
42	VIEW	VW_GBC_22	507K (13)
43	HASH GROUP BY		507K (13)
47	HASH JOIN		63719 (50)
52	TABLE ACCESS FULL	DATE_DIM	95 (33)
54	TABLE ACCESS FULL	CATALOG_SALES	59796 (47)
55	SORT JOIN		180K (5)
59	TABLE ACCESS FULL	CUSTOMER	3589 (31)

Figure 9: TPCDS Q4: GBP (best state) in SEL$1 and SEL$2. Cost = 1872K.

Id	Operation	Name	Cost (%CPU)
0	SELECT STATEMENT		**1875K**(11)
5	UNION-ALL		
10	MERGE JOIN	(SEL$1)	1074K (12)
16	VIEW	VW_GBF_12	180K (5)
17	HASH GROUP BY		180K (5)
18	TABLE ACCESS FULL	CUSTOMER	3589 (31)
19	SORT JOIN		894K (13)
22	VIEW	VW_GBC_11	888K (13)
23	HASH GROUP BY		888K (13)
27	HASH JOIN		88030 (57)
32	TABLE ACCESS FULL	DATE_DIM	95 (33)
34	TABLE ACCESS FULL	STORE_SALES	80280 (52)
39	MERGE JOIN	(SEL$2)	693K (11)
44	VIEW	VW_GBC_23	507K (13)
45	HASH GROUP BY		507K (13)
49	HASH JOIN		63719 (50)
54	TABLE ACCESS FULL	DATE_DIM	95 (33)
56	TABLE ACCESS FULL	CATALOG_SALES	59796 (47)
57	SORT JOIN		180K (5)
61	TABLE ACCESS FULL	CUSTOMER	3589 (31)

Figure 10: TPCDS Q4: GBP in SEL$1($2^{nd}$ best state), SEL$2 when best state of SEL$1 is disabled by hint. Cost = 1875K.

Strategy-2: Fine-Grained Hint - Disable all states

Next, we present the impact on the query plan when all the GBP states are disabled in query block SEL$1. The hint that achieves this effect is NO_PLACE_GROUP_BY(@SEL$1). The corresponding plan is shown in Figure 11. The plan shows that there is no GBP in SEL$1 but is present in SEL$2 as intended. This plan has a cost of 8178K.

Strategy-3: Old Optimizer Version Hint

The impact on the query plan when a global hint that modifies the optimizer version is used, can be observed in Figure 12.

GBP is absent in both the query blocks SEL$1 and SEL$2. Instead, the tables are joined directly. The cost of this plan increases to 32M. When an older optimizer version is used, all the enhancements and bug fixes made after that version are absent, potentially affecting the quality of plans.

The idea behind trying to use fine-grained hints before moving to global hints like optimizer version is to ensure that the quality of plans is least impacted. However, it is possible although less

[1]Strategy hint: NO_PLACE_GROUP_BY(@SEL$1 (STORE_SALES DATE_DIM))
[2]Strategy hint: NO_PLACE_GROUP_BY(@SEL$1)
[3]Strategy hint: OPTIMIZER_FEATURES_ENABLE('19.1.0')

```
--------------------------------------------------------------------
| Id  | Operation                      | Name              | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                   |  8178K (7)|
|   5 |  UNION-ALL                     |                   |           |
|  10 |   MERGE JOIN                   |           (SEL$1) |  1274K (10)|
|  12 |    HASH JOIN                   |                   | 88030  (57)|
|  17 |     TABLE ACCESS FULL          | DATE_DIM          |    95  (33)|
|  19 |     TABLE ACCESS FULL          | STORE_SALES       | 80280  (52)|
|  20 |    SORT JOIN                   |                   |   180K  (5)|
|  24 |     TABLE ACCESS FULL          | CUSTOMER          |  3589  (31)|
|  29 |   MERGE JOIN                   |           (SEL$2) |   693K (11)|
|  34 |    VIEW                        | VW_GBC_11         |   507K (13)|
|  35 |     HASH GROUP BY              |                   |   507K (13)|
|  39 |      HASH JOIN                 |                   | 63719  (50)|
|  44 |       TABLE ACCESS FULL        | DATE_DIM          |    95  (33)|
|  46 |       TABLE ACCESS FULL        | CATALOG_SALES     | 59796  (47)|
|  47 |    SORT JOIN                   |                   |   180K  (5)|
|  51 |     TABLE ACCESS FULL          | CUSTOMER          |  3589  (31)|
```

Figure 11: TPCDS Q4: GBP in SEL$2 when disabled completely in SEL$1 by hint. Cost = 8178K.

```
--------------------------------------------------------------------
| Id  | Operation                      | Name              | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                   |    32M (6)|
|   5 |  UNION-ALL                     |                   |           |
|   6 |   HASH GROUP BY                |                   |    13M  (5)|
|  10 |    MERGE JOIN                  |           (SEL$1) |  3473K  (7)|
|  12 |     HASH JOIN                  |                   |   596K (10)|
|  17 |      TABLE ACCESS FULL         | DATE_DIM          |    88  (28)|
|  19 |      TABLE ACCESS FULL         | STORE_SALES       | 80280  (52)|
|  20 |     SORT JOIN                  |                   |   180K  (5)|
|  24 |      TABLE ACCESS FULL         | CUSTOMER          |  3589  (31)|
|  29 |    MERGE JOIN                  |           (SEL$2) |  1913K  (7)|
|  31 |     HASH JOIN                  |                   |   336K (11)|
|  36 |      TABLE ACCESS FULL         | DATE_DIM          |    88  (28)|
|  38 |      TABLE ACCESS FULL         | CATALOG_SALES     | 59796  (47)|
|  39 |     SORT JOIN                  |                   |   180K  (5)|
|  43 |      TABLE ACCESS FULL         | CUSTOMER          |  3589  (31)|
```

Figure 12: TPCDS Q4: No GBP in SEL$1, SEL$2 when optimizer version 19.1 hint is used. Cost = 32M.

likely, that the reverse is true in certain situations depending on the error and the relationship between different transformations. For example, if the best and the second best states of a transformation in the latest version have errors and the older optimizer version has a best state that is neither of the above, using the older version may be better than disabling the transformation completely on the query block. In our future work we plan to present techniques to deal with such issues by costing plans from different strategies and choosing the best among them.

6.3 A Real-World Survey

We attempt to answer the following questions in order to evaluate the effectiveness of AEM in real-world workloads.

Q1: How common is it for real-world faults to have workarounds?

Q2: How effective is AEM in generating those workarounds? This question leads to three sub-questions that measure the effectiveness of AEM from different aspects.

Q2-1 Success Rate: If a workaround exists, can AEM find it automatically?

Q2-2 Search Time: Is AEM efficient in its searching process?

Q2-3 Quality of Result: If multiple workarounds are available, does AEM choose the best one?

We surveyed Oracle's bug repository for all the compile-time bugs that were reported from customer workloads over a period of one year and could be reproduced in-house.

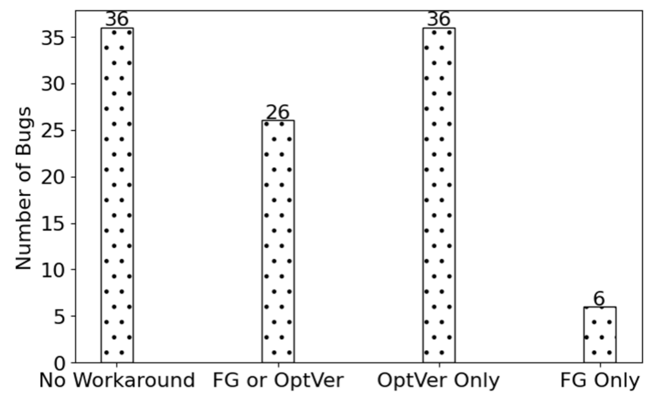

Figure 13: Distribution of compilation-time bugs by the types of workarounds. Four classes are shown i) errors with no feasible workaround ("No Workaround") ii) errors that can be mitigated with fine-grained strategies or optimizer versions ("FG or OptVer") iii) errors that can only be mitigated with optimizer versions ("OptVer Only") and iv) errors that can only be mitigated with fine-grained strategies ("FG Only").

Figure 13 shows among a total of 104 faults, 65.4% (68/104) have some workaround and 34.6% (36/104) have no workaround at all. Among the 68 bugs that have workarounds, 26 can use fine-grained strategies or older optimizer versions. These errors happen during query transformations and can be mitigated by either disabling the transformation or reverting to a version that predates the transformation. Among the rest, 36 bugs do not have fine-grained hint solutions but can be mitigated using optimizer-version strategy. These errors take place in code modules outside the query transformation framework (e.g., physical optimization and code generation). Finally, 6 bugs can only be mitigated by fine-grained strategies because these errors occur in old query transformations that existed prior to the introduction of optimizer versions and hence the transformations cannot be disabled even in the oldest version. However, they can be controlled by fine-grained hints.

To answer Q1 and Q2-1 (discussed above), along the lines of our survey findings, AEM's success rate is close to 65%. From Figure 13, the first category, "no workaround", indicates bugs that do not have workarounds in the form of transformation hints or optimizer versions. As we expand the scope of AEM in future, some of these bugs may benefit from new strategies.

To answer Q2-2, we gathered bugs that could be mitigated by altering optimizer versions and categorized them by version numbers as shown in Figure 14. To create a concise representation, minor versions are lumped together under major version numbers. For example, 5 minor versions are merged into the bar of version 11. When looking for an optimizer version workaround, AEM begins from the most recent version and proceeds to older versions using every minor version in between. For example, if the current version is 20, AEM starts from version 19 and proceeds to older versions.

Figure 14 shows that 61.3% (38/62) of bugs can be mitigated by the 10 most recent optimizer versions, corresponding to the first 28.6% (10/35) of versions, and 87.1% (54/62) of the bugs can be mitigated

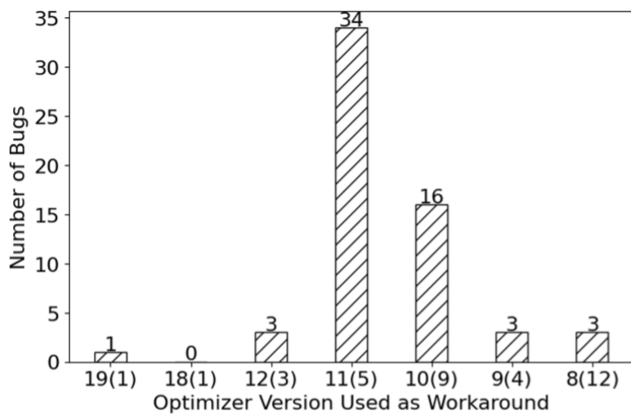

Figure 14: Distribution of compile-time bugs that can be mitigated by changing optimizer versions. Inside parentheses is the number of minor versions of a major version. For example, version 11 has 5 minor versions and is displayed as 11(5).

by exploring the first 54.3% (19/35) of versions. We observed a disproportionate concentration of bugs on newer releases, which is expected as new features are introduced in them that stabilize over time. In response to Q2-2, this finding confirms that it is efficient to search from newer optimizer versions to older ones.

To answer Q2-3, we need to compare the true performance of plans generated by different workarounds in real-world customer workloads. Currently we have limited data in this regard as AEM hasn't been released in the market. In addition, many historical bugs reported by customers are difficult to reproduce in-house. Nevertheless, we plan to collect extensive feedback on AEM and its impact on real-world query performance in the next release of Oracle database.

We can cite empirical evidence from our experience that most customers have been satisfied with manual workarounds provided to them. AEM uses more sophisticated strategies leading us to believe the quality of workarounds can meet the needs of most users.

7 RELATED WORK

The growing scale and complexity of modern software have made it an increasingly challenging task for developers to deliver timely fixes and workarounds to software bugs. Naturally, many researchers and practitioners turned towards automatic software repair. Over the past few decades, the field garnered a lot of research on self-healing and self-repairing techniques for myriad scenarios of software applications [15, 19, 23].

Automatic repairing techniques can be categorized into two families by their domain of use. General solutions tackle programming errors that are common in software applications; examples are buffer overflow [27], infinite loops [7] and inconsistent data structures [13], etc. Domain-specific solutions exploit the characteristics of an application to devise efficient solutions. For example, Gopinath et al. [16] proposed a technique to fixing bugs in a SQL-like proprietary language called ABAP. Carzaniga et al. [9] focused on web applications and presented a solution to component failures caused by faults in popular access libraries. The AEM framework proposed in this paper is also a domain solution - specific to the context of SQL compilation. To our knowledge, our work is the first to apply automatic healing techniques to SQL engines and devise SQL-tailored mitigation solutions.

AEM mitigates errors by applying strategies that lead query compilation through alternative code paths that may bypass faulty conditions. This idea exploits a classical concept in fault-tolerant software engineering, known as the multi-version programming [3]. Error recovery through the multi-version approach can only be achieved in software applications with in-built design diversity [4]. In such systems, the same functions are implemented with different designs and multiple instantiations. On encountering failure, execution restores to a common starting point and switches to an alternative implementation [26]. The multi-version approach has proved its efficacy in many real-world scenarios, with the switching of implementations taking place at various levels of granularity, such as replacement of buggy API calls, re-arrangement of operation sequences [8, 9, 17], or switching between multiple versions of the same application run in parallel [18].

Unlike the models of the above research work, AEM has a unique ability to exert fine control on the granularity of alternative implementations. It is possible to switch to a different implementation of SQL compilation at the plan level, query block level, or the statement level (a statement has multiple plans). This fine control is achieved by tracking the query compilation context and using semantically flexible SQL hints. Plan-level failover is favored over query-block-level hints as historical plans usually have a proven record of successful execution and acceptable performance. Query-block-level hints are in turn favored over statement-level hints because the former lead to localized workarounds that minimize divergence from the error-free query plans.

Learned query optimizers [20, 21] have been proposed to predict and correct performance of queries in databases. They work on top of existing query optimizers and generate corrective actions to fix performance issues in query plans. Bao [20] engine uses hints to learn how plans respond to certain actions and converges on promising actions specific to each query. In the context of error mitigation, we observe that learning techniques can be used to i) predict when queries may encounter errors or ii) predict promising corrective actions. Applying the techniques discussed in [20, 21] to the former is tricky as the rate of errors is typically low and the overhead to compilation time may be unacceptable. Also, due to the unpredictable nature of errors, it may be hard to identify the relevant features and attributes. Our model takes a reactive approach of mitigating errors when they occur. Identifying promising corrective actions based on historical error-mitigation data, on the other hand, can benefit from learning techniques and we plan to utilize such them as we expand the scope of AEM and improve its turnaround time. In this context, techniques proposed by Bao may potentially be used to achieve similar effect, although using a reactive approach.

Most SQL autonomous features focus on performance tuning [11], addressing performance regressions through automating the creation of physical structures like indexes [12, 24], materialized views [1] and partitioning schemes [10], maintaining plan stability

as in the Automatic Plan Correction of SQL Server [22] and the Automatic SQL Plan Management of Oracle [5, 29], and searching for an optimized set of system configuration knobs [28]. Unlike those works, AEM handles crashing SQLs instead of slow-running SQLs. But the methods of AEM share some overlap with the common practices used in performance tuning, like switching to alternate plans and modifying system configurations. The experimental results of AEM show that these approaches traditionally used in performance tuning can also find application in error recovery and fault tolerance.

8 CONCLUSION AND FUTURE WORK

In this paper, we explored automatic error mitigation in the context of a database' online user process (foreground) and presented various strategies to transparently mitigate errors encountered during SQL query compilation. We showed how the strategies can be less disruptive to the quality of query plans when mitigating errors thus creating a better user experience.

Currently AEM stops as soon as it identifies a workaround – this can be seen as a **"First Fit"** model. In the future, we plan to enhance AEM to consider plan quality while mitigating errors. While this can be performed online, it is best done offline by a background process of the database that periodically checks whether new compilation-errors incidents occurred and explores effective mitigation strategies. A background process has a relatively larger time budget to find a workaround and can explore more candidate strategies, cost them and compare the quality of plans before finalizing a mitigation strategy forming what can be termed as a **"Best Fit"** model. It can also complete the work of online foreground processes that time out before identifying a workaround.

We also plan to extend AEM to i) cover a wider range of compilation errors (e.g., physical optimizer errors) ii) use a wider range of candidate hints for mitigation (e.g., more parameter controls, transformations that were applied prior to the active one at the time of error etc.) iii) recover from query execution errors. As the query compilation context has information on the sequence of operations until the time of error, candidate generation can be improved to consider prior transformations in addition to the active transformation. And as mentioned earlier, error mitigation at execution time is more challenging as cleanup and restart of query execution is required.

ACKNOWLEDGMENTS

We would like to thank Sunil Chakkappen and Palash Sharma for their contribution and valuable feedback in designing this feature.

REFERENCES

[1] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated Generation of Materialized Views in Oracle. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3046–3058. https://doi.org/10.14778/3415478.3415533

[2] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-Based Query Transformation in Oracle. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1026–1036.

[3] A. Avizienis. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering* SE-11, 12 (1985), 1491–1501. https://doi.org/10.1109/TSE.1985.231893

[4] Benoit Baudry and Martin Monperrus. 2015. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Comput. Surv.* 48, 1, Article 16 (Sept. 2015), 26 pages. https://doi.org/10.1145/2807593

[5] Nigel Bayliss. 2019. *Automatic tuning*. Oracle. Retrieved July 7, 2023 from https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-sql-plan-mgmt-19c-5324207.pdf

[6] Nigel Bayliss. 2020. *What is the Automatic SQL Tuning Set?* Oracle. Retrieved July 7, 2023 from https://blogs.oracle.com/optimizer/post/what-is-the-automatic-sql-tuning-set

[7] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 609–633.

[8] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. 2013. Automatic recovery from runtime failures. In *2013 35th International Conference on Software Engineering (ICSE)*. 782–791. https://doi.org/10.1109/ICSE.2013.6606624

[9] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. 2010. Automatic Workarounds for Web Applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 237–246. https://doi.org/10.1145/1882291.1882327

[10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[11] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 1098–1109.

[12] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 666–679. https://doi.org/10.1145/3299869.3314035

[13] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. 2007. Assertion-Based Repair of Complex Data Structures. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 64–73. https://doi.org/10.1145/1321631.1321643

[14] Kurt Engeleiter. 2009. *Using Automatic Workload Repository for Database Tuning: Tips for Expert DBAs.* Oracle. Retrieved July 7, 2023 from https://www.oracle.com/technetwork/database/manageability/diag-pack-ow09-133950.pdf

[15] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[16] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. 2014. Data-Guided Repair of Selection Statements. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 243–253. https://doi.org/10.1145/2568225.2568303

[17] Alessandra Gorla, Mauro Pezzè, Jochen Wuttke, Leonardo Mariani, and Fabrizio Pastore. 2012. Achieving Cost-Effective Software Reliability Through Self-Healing. *COMPUTING AND INFORMATICS* 29, 1 (Jan. 2012), 93–115. https://www.cai.sk/ojs/index.php/cai/article/view/75

[18] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *2013 35th International Conference on Software Engineering (ICSE)*. 612–621. https://doi.org/10.1109/ICSE.2013.6606607

[19] Angelos D. Keromytis. 2007. Characterizing Software Self-healing Systems. In *Computer Network Security*, Vladimir Gorodetsky, Igor Kotenko, and Victor A. Skormin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–33.

[20] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.* 51, 1 (June 2022), 6–13. https://doi.org/10.1145/3542700.3542703

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[22] Microsoft. 2023. *Automatic tuning*. Microsoft. Retrieved July 7, 2023 from https://learn.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning?view=sql-server-ver16

[23] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[24] Arup Nanda. 2021. *Automatic indexing with Oracle Database.* Oracle. Retrieved July 7, 2023 from https://www.oracle.com/news/connect/oracle-database-automatic-indexing.html

[25] Krishna Kantikiran Pasupuleti, Dinesh Das, Satyanarayana R Valluri, and Mohamed Zait. 2022. Observability of SQL Hints in Oracle. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta,

GA, USA) *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 3441–3450. https://doi.org/10.1145/3511808.3557124

[26] Brian Randell. 1975. System structure for software fault tolerance. *IEEE Transactions on Software Engineering* SE-1, 2 (1975), 220–232. https://doi.org/10.1109/TSE.1975.6312842

[27] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 124–135. https://doi.org/10.1109/DSN.2014.25

[28] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[29] Mohamed Ziauddin, Dinesh Das, Hong Su, Yali Zhu, and Khaled Yagoub. 2008. Optimizer Plan Change Management: Improved Stability and Performance in Oracle 11g. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1346–1355. https://doi.org/10.14778/1454159.1454175