IBM Research Report

Data-Guided Repair of Selection Statements

Divya Gopinath
University of Texas at Austin

Sarfraz Khurshid
University of Texas at Austin

Diptikalyan Saha
IBM Research - India

Satish Chandra
IBM Research - India

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.
Abstract

Database-centric programs form the backbone of many enterprise systems. Fixing defects in such programs takes much human effort due to the interplay between imperative code and database-centric logic. This paper presents a novel data-driven approach for automated fixing of bugs in the selection condition of database statements (e.g., \texttt{WHERE} clause of \texttt{SELECT} statements) – a common form of bugs in such programs.

Our key observation is that in real-world data, there is information latent in the distribution of data that can be useful to repair the selection condition efficiently. Given a faulty database program and input data, only a part of which induces the defect, our novelty is in determining the correct behavior for the defect-inducing data by taking advantage of the information revealed by the rest of the data. We accomplish this by employing semi-supervised learning to predict the correct behavior for defect-inducing data and by patching up any inaccuracies in the prediction by a SAT-based combinatorial search. Next, we learn a compact decision tree for the correct behavior, including the correct behavior on the defect-inducing data. This decision tree suggests a plausible fix to the selection condition.

We evaluate our approach on a suite of programs and data sets obtained from real-world applications. The evaluation shows the effectiveness of our approach.

1 Introduction

A majority of enterprise software systems are database-centric programs. Defects in such programs, specifically in database manipulating statements, are expensive to fix and can require much human effort in understanding the interplay between traditional imperative code and database-centric logic. Automated tools to help diagnose these defects, and furthermore, to assist with fixing them can make a substantial reduction in the cost of developing and maintaining database-centric programs.

1.1 Problem Context

Our specific focus is on SAP ERP systems, in which database-centric programming is carried out in a proprietary language called ABAP. ABAP contains SQL-like commands, but it mixes imperative code and SQL’s declarative syntax. We introduce the essential constructs of ABAP that are relevant for this paper using a small example (Figure 1).

The meaning of this ABAP code segment is straightforward. At line 1, it reads all rows from a database called \texttt{OrderTab} into an internal table called \texttt{itab}. The \texttt{SORT} statement sorts this internal table by \texttt{CstId}, which is the key. The \texttt{DEL} statement at line 4 removes from \texttt{itab} those rows that match the condition described in the statement. The \texttt{LOOP} at line 5 iterates over \texttt{itab}. When it encounters a new \texttt{CstId}—that is when \texttt{NEW} at line 6 is true— it resets an accumulator called \texttt{amount}, and it prints the accumulated amount when the last record of that

```
1 SELECT CstId Price Year from OrderTab INTO itab
2 3 SORT itab by CstId
4 DEL from itab where Year <= 2009 and Price > 5
5 LOOP AT itab INTO wa
6 7 AT NEW CstId
8 amount=0
9 ENDAT
10 amount = amount + wa.Price
11 END
12 WRITE wa.CstId amount
13 ENDLOOP
```

Figure 1: A sample ABAP code segment.
CstId has been visited; this is done when AT END on line 10 is true. (AT NEW and AT END help with key-wise aggregation akin to the SQL GROUP-BY construct.)

<table>
<thead>
<tr>
<th>CostId</th>
<th>Price</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>2012</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>2011</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>2001</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>2002</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>2011</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2005</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>2007</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>2010</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2011</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2012</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2009</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>2001</td>
</tr>
</tbody>
</table>

Table 1: A sample input to program in Figure 1. The last column with ‘+’/’-’ is not a part of the input table.

Suppose the program in Figure 1 is run on the database in Table 1. The rows marked ‘+’ are retained in itab after the DEL statement. The output of the program is, which unfortunately differs from the expect output, also shown:

<table>
<thead>
<tr>
<th>ID</th>
<th>Amount</th>
<th>Expected Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

The bug arises from an error in the condition of the DEL statement, which causes the first row for CstId 2 to be incorrectly deleted (shown by a bold ‘-’).

We can think of the DEL statement as a (equivalent) SELECT statement: SELECT * FROM itab WHERE Year > 2009 OR Price <= 5. We call such a defect a selection bug, because the bug is due to an incorrect WHERE condition in a SELECT statement. The problem is to find an alternate WHERE condition for the faulty SELECT statement (whose location is assumed to be known), so that the entire output, corresponding to each of the keys, is correct.

Selection bugs are common in ABAP programs. In fact, many database statements in ABAP program allow a selection condition, and therefore, are vulnerable to a selection bug. For example, the ABAP READ and DELETE ADJACENT statements can be modeled as selection statements. Based on our experience working with practitioners in IBM Global Business Services, about 25% of the ABAP code level defects have to do with selection. Such bugs typically do not reveal themselves while testing with limited set of data that is available in the test environment. The production environment has a lot more data and therefore exposes the corner cases that do not show up while testing. Moreover, the lack of an automated test-case generation tool for this framework is another reason why such bugs are not discovered while testing. Therefore, techniques that can help in fixing defective selection statements are of much value.

Note that in our setting of debugging ABAP programs, the process starts with the end user of this software filing a bug report, citing a deviation of actual output from the expected output on given input data. Thus, the expected output of the program is already known to the programmer (or the maintainer). As we shall see, the challenge here is in determining the correct behavior of the defective SELECT statement from the expected output of the entire program, and in determining an alternate WHERE condition for the selection that would match the correct behavior.

An obvious technique to generate a correct selection condition would be to explore the space of syntactic mutations of the buggy condition. Because of the possible presence of data values in the
clauses that constitute the conditions, the search space for a mutation-based technique is immense. The size of the mutation search space for our suite of benchmarks is reported in Section 3. This makes the technique very inefficient for real use. In comparison, our approach, presented next, completely sidesteps the drawbacks of a mutation-based approach.

1.2 A Data-Driven Approach: an Overview

Our key observation is that in real-world data, there is information latent in the distribution of data that can be useful to repair the WHERE condition efficiently. Syntactic search completely ignores this latent information. In this work, we show that it is possible to find a repair suggestion efficiently if we took advantage of this information.

In our approach, we first discover the correct behavior of the selection statement on the failure causing input data, and then find an alternate selection statement that exhibits the correct behavior. Our approach leverages the distribution of input data in both of these phases.

A defective selection statement assigns incorrect + or - labels to some of the rows of the input; for example, some of the rows for CstId 2 do not have the correct labels. To discover the correct behavior of the defective selection statement, we need to search through all possible assignments of labels to rows that have possible incorrect labels. Our technique carries out this search efficiently by taking advantage of the distribution of data.

Since part of the output is correct, we know that the rows that contributed to that part are labeled correctly; the remaining rows are possibly mislabeled. Our premise is that a possibly-mislabeled row that is geometrically close to a correctly-labeled row is likely to require the same label. Obviously, this notion of proximity is not guaranteed to produce the correct labels, but they can serve as a very good starting point from which to carry out the search for the right labeling. This is exactly what we do: use a labeling computed on the basis of geometric proximity, but fix it up based on local search around that labeling.

In Figure 2, data of passing keys in Table 1 is shown with diamonds for positively labeled data and squares for negatively labeled data. For rows belonging to CstId 2, whose labels as generated by the WHERE condition are suspect, data is shown with a triangle (unlabeled). Assuming that points that are spatially close are likely to be labeled similarly, an assignment of a positive label to the two unlabeled points on the right can be done with relatively high confidence. The two unlabeled data points in the middle of the chart could go either way, so an assignment of a negative label to the two points in the middle can be done only with low confidence. Table 2 shows a sample assignment of predicted labels to the failing rows. We generated these predications using an implementation of support vector machines (SVM [23, 2]). Informally, SVM creates separating lines A, B, and their center–C–as its best effort on how to separate positively and negatively labeled data. In this particular example, the line D would have be the perfect separator; so, two of the points that are close to SVM’s separator (C) are predicted incorrectly.

<table>
<thead>
<tr>
<th>CstId</th>
<th>Price</th>
<th>Year</th>
<th>Predicted Label</th>
<th>Correct Label</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>2005</td>
<td>+</td>
<td>+</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>2007</td>
<td>+</td>
<td>-</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>2010</td>
<td>+</td>
<td>+</td>
<td>0.9</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2011</td>
<td>+</td>
<td>+</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 2: Predicted label assignment for the failing rows

The incorrect lower-confidence predictions can be fixed up by a combinatorial search for labels,
until we obtain a correct label for all the rows (correctness can be validated by the final assertion.) We carry out this search iteratively, starting with the row with the least confident prediction. In realistic problem sizes, this strategy—which takes advantage of data distribution—is significantly more efficient than combinatorial search on all the rows.

Once we have the correct labels for all the rows, the problem reduces to that of finding a function (a classifier) that attaches correct + or - labels to rows depending on the contents of the row. As mentioned before, it is difficult to find such a function by looking for syntactic variations on the existing WHERE condition.

In general, a vast number of different functions could produce the correct labels for a given set of rows. Not all of these would be close to the one intended by the programmer, because they may be overfitted to the data, in the sense that those functions may not label as-yet-unseen data correctly. A common heuristic is to look for a compact function, because it is more likely to be generalizable, and therefore (presumably) correct.

We use the distribution of data to guide the search for a compact function using a well-known technique known as decision-tree learning. This technique performs a greedy search over a space of functions, being guided by the distribution of data and the label for each row of data. The way this greedy search works is to first identify a predicate that classifies most of the data correctly, and then iteratively identify additional predicates as required to classify the residual data.

In the running example, first it would realize that partitioning the rows of the table on the basis

![Figure 2: Distribution of data in Table 1](image-url)
Figure 3: Splitting the data points on the basis of alternative conditions

of $Year \leq 2008$ gives the maximum, though not perfect, efficacy in terms of clubbing the + and - labeled rows in distinct partitions. See Figure 3(a), which shows the result of splitting on the basis of $Year \leq 2008$; again, squares are negatively labeled points and diamonds are positively labeled points, and the data is for the correct labels on all rows of Table 1. An alternate split, say on the basis of $Price \leq 10$, shown in Fig 3(b) is less effective in clubbing the + and - labeled data in distinct partitions. The ID3 decision-tree learning algorithm [20] captures this intuition using the concept of information entropy and automatically chooses the most advantageous splitting predicate.

In the partition for which $Year \leq 2008$, the maximum efficacy is obtained by further partitioning on the basis of $Price \leq 8$, at which point, all positives are perfectly separated from the negatives. In the partition for which $Year > 2008$, all rows are positive regardless of the price. This decision tree is (written as conjunctions of clauses on paths from root to +ve leaf nodes, and disjunction over such paths): $Year > 2008 \lor (Year \leq 2008 \land Price \leq 8)$. By DeMorgan’s laws, this simplifies to the following condition: $Year > 2008 \lor Price \leq 8$

Comparing this to the previous incorrect WHERE condition, we see that while the learned WHERE clause for $Year$ is slightly but gratuitously different, for $Price$ it is crucially different.

Our technique manages to find “natural” conditions that a programmer would have written, and therefore ends up offering useful repair suggestions. We attribute this desirable property to our data-guided approach. A WHERE condition that a programmer writes is intended to classify regions of data uniformly, as opposed to cherry picking points in the data space and classifying them individually by some complex conditional logic. This is the reason that predictions based on spatial proximity work quite well, and this is also the reason that the heuristic of finding a compact decision tree works.

1.3 Contributions

Our contributions are the following:

1. We describe a new approach for repairing faulty selection conditions in database statements. Our approach tries to extract information from the way the incumbent selection condition treats
Repair (s, Out, CorrectOut, key_fields, Trace)
// initialize
find PKeys, FKeys, sPKeys, sFKeys
// step 1: predict labels for failing keys input of s
projPrediction, inputPrediction, Map, prodTbl =
predict(sFKeys, sPKeys, sIn, sOut, s)
// step 2: Correct label computation
sCorrectOutSet = label(s, Trace, sFKeys, projPrediction)
// step 3: WHERE condition generation
generate_conditions(sCorrectOutSet, prodTbl, Map, inputPrediction)

Figure 4: Algorithm: Generating repair suggestions for selection statement

the part of the data that exhibits correct behavior, and from the relative distribution of passing and
defect-inducting data.

2. We give a new way of combining machine learning and combinatorial search in determining
the correct labels for the failing keys. The learning part takes advantage of the known behavior of
the passing keys, whereas, the combinatorial part makes up for cases in which the knowledge for
passing keys does not extend perfectly to the failing keys.

3. We present an evaluation of the proposed approach on a suite of programs drawn from an
industrial setting. These programs are excerpts of real programs, and the data sets we use come
from real data: this is crucial because the effectiveness of our approach cannot be gauged based on
synthetic data, which may not be representative of distributions found in real data. The evaluation
indicates promise in the approach.

2 Repair Algorithm

In this section we describe the repair algorithm in detail. As shown in Figure 4, the algorithm has
three major steps: 1) exploit distribution of data to predict the selection result for the input rows
of the faulty selection statement, 2) verify the prediction and if required determine the selection
result using combinatorial search for the parts where the predictions are incorrect, and 3) generate
correct conditions using an existing decision-tree learning algorithm.

We illustrate the various steps of the algorithm using an example shown in Figure 5. The
example is a slight variation of the example presented in Section 1 to illustrate some salient features
of the algorithm. The SELECT statement (shown in Lines 1-4) is the faulty statement. Below are
the entities used in the algorithm:

- Trace: the execution trace which produces incorrect output.
- s: the trace occurrence of the faulty statement
- sIn, sOut: the set of input tables and current output of s
- Out: incorrect program output
- CorrectOut: expected correct program output
- sCorrectOut: a correct output of s which can produce CorrectOut
- key_fields: a set of fields which uniquely identifies each row of Out
```sql
SELECT CstId Price Year
from Order, Material into itab
where Item = ItemId
and Year > 2009
LOOP AT itab INTO wa
  amount = 0
ENDAT
  amount = amount + wa.Price
ENDLOOP
```

**Program Output (Out)**

<table>
<thead>
<tr>
<th>CstId</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
</tr>
</tbody>
</table>

**Correct Program Output (CorrectOut)**

<table>
<thead>
<tr>
<th>CstId</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
</tr>
</tbody>
</table>

**Figure 5:** An ABAP program and data; s is the SELECT statement

The `Out`, `CorrectOut`, `sIn`, `sOut`, `sCorrectOut` for the example are shown in Figure 5.

In the domain of data-centric programs, each row in the output is identified by a set of field-value pairs, called `key`. The algorithm compares the current program output (`Out`) and expected output (`CorrectOut`) to determine a set of failing keys (`FKeys`) and passing keys (`PKeys`) for the program output. A passing key is a set of key field-value pairs which identifies identical rows in `Out` and `CorrectOut`. A failing key is a set of key field-value pairs which identifies a row which exists in `Out` but not in `CorrectOut` or vice versa or identifies a row in `Out` and a row in `CorrectOut` which differ in at least one non-key field value. In the example, `CstId=3` is the passing key, whereas `CstId=1` and `CstId=2` are the failing keys. Note that, `CstId=1` corresponds to a missing row in the output.

As described in Section 1, the prediction algorithm predicts the selection result of the failing input rows using the selection result for the passing input rows. All the input rows in `s`, obtained using Cartesian product of all tables in `sIn`, that directly or indirectly affect the failing rows (incorrect/missing/unwanted) in the program output are failing rows in the input of `s`, everything else are passing rows. Such classification is performed based on key-based dependency from the passing and failing keys in the output of the program, as this will track the un-selected rows and the missing output rows. The key-based dependency finds failing and passing rows in the input. In the example, our algorithm finds failing input rows corresponding to failing keys `(CstId, 1)` and `(CstId, 2)`, and passing input rows corresponding to passing key `(CstId, 3)`. The set of passing and failing keys for `s` are denoted as `sPKeys` and `sFKeys`.

**Prediction of correct output of s** The Function `predict` in Figure 6 first creates the input (`prodTbl`) by performing Cartesian product of all input tables (in the example, `Material` and `Order`), then assigns label 0 (signifying rows whose value to be predicted) to the input rows corresponding to
the failing keys. The current selection clause behaves correctly (either selects or deselects) for the passing rows. Each passing row in the input is thus classified as +1 if the row is selected, and -1 if the row is not selected, denoting known and correct classification (Lines 7-11). This forms an input to an SVM, which classifies each row with label 0 with a signed value called prediction (Line 14). The sign (called Label) predicts whether the input row is to be selected (positive value) or not selected (negative value). The unsigned prediction value denotes the confidence associated with the prediction. In the example, there are 121 rows in the input prodTbl, 22 of them are for CstId=1 and 44 rows are for CstId=2 which are marked 0, whereas the remaining 55 rows for CstId=3 are labeled +1 (22 rows) or -1 (33 rows).

Note that a SELECT statement first performs the selection of the rows of prodTbl, followed by projection to generate the output of $s$. A projected row is formed after the projection of an input row. The set of input rows which projects to a projected row, is called the block of the projected row. A projected row is in output of $s$, if at-least one input row from its block is selected. SVM based prediction only predicts the outcome of the selection. Using such prediction, our algorithm determines the prediction for the projected rows (Lines 18-22). The likelihood of selection of a projected row is determined by the maximum likelihood of selection of the input rows in its block. The algorithm determines the prediction of the projected row as the maximum prediction (signed value) of all the rows in its block.

In the example, there are 121 rows in the prodTbl, which project to 33 rows, 11 rows per key. The predictions for the projected rows of CstId=1 and CstId=2, as determined by our algorithm, are shown below.
label(s, Trace, sFKeys, prediction)
model = code2model(Trace,s)
sCorrectOutSet = empty
for each sFKey ∈ sFKeys
  threshold = 0
while(true)
  label = empty
  for each p ∈ prediction.KeySet s.t. p.key=sFKey
    predValue = prediction.get(p)
    if |predValue| > threshold
      label(p)=true if predValue > 0
      label(p)=false if predValue < 0
    keyCorrectOutSet = SAT(fixedInp(sIn,label) U unknownInp)
    A model ∧ CorrectOut
    where unknownInp ⊆ {sIn – fixedInp}
    if keyCorrectOutSet is Empty
      threshold = threshold + ParamThresholdIncrement
      return failure
    else
      continue;
    else
      break
  sCorrectOutSet = sCorrectOutSet X keyCorrectOutSet
  Add passing key data to each selection in sCorrectOutSet
return sCorrectOutSet

Figure 7: Algorithm: Labeling for failing keys

<table>
<thead>
<tr>
<th>CstId</th>
<th>Price</th>
<th>Year</th>
<th>Pred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2009</td>
<td>+0.8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>2002</td>
<td>-0.4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2012</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>2005</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>2006</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>2007</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>2011</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>2012</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>2001</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>2001</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>2002</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CstId</th>
<th>Price</th>
<th>Year</th>
<th>Pred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>2009</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2002</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2012</td>
<td>+1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2005</td>
<td>-0.1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2006</td>
<td>+0.1</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>2007</td>
<td>+0.2</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>2011</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>2012</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>2001</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>2001</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2002</td>
<td>-1</td>
</tr>
</tbody>
</table>

The salient features of this step are summarized here:

- Determining passing and failing input rows and exploiting the data-distribution to predict selection result of the failing input rows.
- Mapping prediction values from input rows to the projected rows based on maximum likelihood of selection.

Determining Correct Output of s The next step is to use these signed values to determine the part of sCorrectOut corresponding to the failing keys. The algorithm, described in Figure 7, tries to determine this per failing key (Line 4), for a reason described later. The decision whether a projected row can be labeled by prediction is done based on its confidence (Lines 8-12). A parameter threshold is used to gradually un-label low confidence projected rows if predicted labeling does not yield an answer. A projected row having confidence value above this threshold is labeled. The label is determined by the sign of prediction, which if positive denotes that the row is to be selected to the output, and not selected (label = false) if negative. Note that, the algorithm starts with threshold value zero to make all the projected rows labeled. In the running example, for CstId=1, the prediction selects only one row with Price=10 (corresponding to item i1),
whereas selects 3 rows with Price values 5 (i3), 7 (i5), and 14 (i6).

Next, we discuss how we verify whether the predictions based labeling yields $s_{CorrectOut}$ and if not, use combinatorial search to label the rows whose predictions are relaxed based on the threshold adjustment.

If $s$ contains a selection bug, then there exists a subset of its projected rows that can produce the expected program output. Such correct selection is already available for the projected rows corresponding to the passing keys. For all the projected rows corresponding to the failing keys, such correct selection can be obtained using a combinatorial search. The scalability of the search depends on the input table size, which is usually large, even if its performed per failing key.

We exploit the labeling predicted for the all or part of projected failing key rows in the previously discussed step and perform combinatorial search in the space of rest of the projected rows corresponding to each failing key. In the first iteration, when threshold=0, no search is required to validate the predicted labels for all projected failing key rows.

The algorithm creates a model of the code which can execute after $s$ as a first order formula using Alloy (Line 2). The details of the translation is provided in Figure 13.

A SAT solver (Line 14) is used to validate the predicted labels or to perform combinatorial search. This is done by invoking SAT using the following formula, $\text{fixedInp}(sIn, label) \cup \text{unknownInp} \land \text{model} \land \text{CorrectOut}$. $\text{fixedInp}(sIn, label)$ represents a set which contains only those rows in sIn for which the label has been predicted to be positive and has confidence greater than the running threshold. A satisfiable solution to the formula is a solution to $\text{unknownInp}$, i.e. a subset of unlabeled records which when combined with the remaining records for that failing key leads to a state that yields the CorrectOut. Model acts as a transfer function in first order logic which translates its input state to the final output state of the program. The SAT invocation either returns no solution or returns a set of correct selections per failing key.

The SAT solver may not yield any satisfying truth assignments to the unlabeled projected rows for a failing key. This is due to the incorrect label prediction done to the projected rows, as, such prediction, in combination with any subset of the unlabeled projected rows may not generate the program output. In this case, the algorithm increases the threshold to un-label some more low confident projected rows. Iteration based on adjusting threshold increases the domain size for combinatorial search. However, in our experiments we have seen that many iterations are seldom needed.

In the running example, SAT solver validates the prediction for CstId=1, but fails for CstId=2 (yields total 26, expected 19). The low confidence projected rows for CstId=2, corresponding to ItemId=i4,i5,i6 are unlabeled. Combinatorial search in the second iteration yields two satisfactory combinations for CstId=2 - (i3, i4, i5), and (i3, i6). Both of them yield total Price value 19 as in CorrectOut.

Combining all solutions for failing keys (Line 26) and adding the passing key selection (Line 27) the algorithm generates the following two $s_{CorrectOut}$.

<table>
<thead>
<tr>
<th>CstId</th>
<th>Price</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2009</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2012</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>2007</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>2011</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>2012</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CstId</th>
<th>Price</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2009</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2012</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2005</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2006</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>2011</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>2012</td>
</tr>
</tbody>
</table>
```python
label(s, Trace, sFKeys, prediction)
model = code2model(Trace, s)
sCorrectOutSet = empty
for each sFKey ∈ sFKeys
  thresholdP = 0
  thresholdN = 0
while(true)
  label = empty
  for each p ∈ prediction.KeySet s.t. p.key=sFKey
    predValue = prediction.get(p)
    if predValue > 0
      if [predValue]> thresholdP cntP++, labelP.add(p)
    else
      if [predValue]< thresholdN cntN++, labelN.add(p)
    if ((cntP + cntN) < cnt_thres)
      for each r ∈ labelP
        labelP.(r) = true
        for each r ∈ labelN
          labelN.(r) = false
      if ((cntP < cnt_thres)
        for each r ∈ labelP
          labelP.(r) = true
        else
          if ((cntN < cnt_thres)
            for each r ∈ labelN
              labelN.(r) = false
      keyCorrectOutSet = SAT(fixedInp(sIn.label) U unknownInp)
      A model ^ CorrectOut)
      where unknownInp ⊆ {sIn - fixedInp}
    if keyCorrectOutSet is Empty
      thresholdP = thresholdP + ParamThresholdIncrementP
      thresholdN = thresholdN + ParamThresholdIncrementN
    if(thresholdP and thresholdN > maximum_confidence_value)
      return failure
    else
      continue;
    else
      break
  sCorrectOutSet = sCorrectOutSet X keyCorrectOutSet
Add passing key data to each selection in sCorrectOutSet
return sCorrectOutSet
```

Figure 8: Algorithm: Optimized Labeling for failing keys

Note that, there are two advantages of performing the iteration per failing key. Firstly, it reduces the space of the combinatorial search from all projected rows to projected rows per key. Secondly, the iterative threshold relaxation can be done only for those rows corresponding to a failing key where prediction was not accurate, thus avoiding un-necessary un-labeling of other failing key rows.

**Optimized threshold relaxation for scalability.** In some cases, relaxing rows based on the threshold adjustment using ParamThresholdIncrement, may unlabel a large number of rows, which in turn may choke the SAT solver. We adopt an optimized approach to improve the scalability. The basic approach uses a common value of ParamThresholdIncrement to increase the threshold for both positively and negatively predicted rows. However, in some cases, when there is an uneven distribution of positively and negatively labeled rows in the passing key data (training data for SVM), the predictions for positive and negative labels are also un-even. For instance, when there are a large number of negatively labeled rows and very few positively labeled rows (mostly seen in joins), the predictions for negative labels have more confidence than positive labels. The negative prediction values are also more precise and evenly distributed as against positive prediction values, where many rows may be assigned the same low confidence prediction. In such a case, un-labeling
generate_conditions(sCorrectOutSet, prodTbl, Map, inputPred)

ConditionSet = empty
for each sCorrectOut ∈ sCorrectOutSet
    weight(sCorrectOut) = average prediction value of sKey rows in answer
rankedSet = sort sCorrectOutSet based on weights
for each sCorrectOut ∈ rankedSet in order
    c = getCondition(prodTbl, sCorrectOut, Map, prediction)
    ConditionSet.add(c)
if no more results required
    return ConditionSet;

getCondition(prodTbl, sCorrectOut, Map, predictedlabels)
for each projectedRow p ∈ Map.keySet
    if p ∈ sCorrectOut
        max_pred = max. prediction of rows in Map.get(p)
        for each r ∈ Map.get(p)
            r_pred = predictedlabels.get(r)
            class(r) = +1, if r_pred = max_pred
            or |r_pred|>threshold, r_pred > 0
            = -1, if |r_pred|>threshold, r_pred < 0
    else
        for each r ∈ Map.get(p)
            class(r) = -1
return ID3(class)

Figure 9: Algorithm: Selection condition generation

of rows based on a common value for ParamThresholdIncrement for both positive and negative predictions, may bring in a large number of positively predicted rows vs. the negatively predicted ones. We adopt a staged approach, where we maintain a ParamThresholdIncrement for the set of positive predictions and another for the set of negative predictions, and relax only one of the sets (whichever would unlabeled lesser number of rows). If this does not yield satisfiability, we relax the other set as well (refer Figure 8).

The salient features of this step are summarized below:

- Performing per-failing key based search and prediction to reduce search space of combinatorial search to determine the set of correct statement output.
- An iterative algorithm to label fewer projected rows based on predictions, if the current labeling does not yield sCorrectOut.

Selection condition generation  The algorithm for selection condition generation is presented in Figure 9. The generation of selection condition is performed using a home-grown implementation of the ID3 decision-tree learning algorithm which learns a classifier that provides 100% accurate classification for the training data. The Function ID3 takes selection result for the input rows to derive at a compact condition satisfying the selection. The algorithm first ranks the set of correct outputs of s derived at the previous step and calls the function getCondition corresponding to each sCorrectOut in order (Lines 7-11).

The Function getCondition classifies each input row of s with +1 or -1 based on the labeling obtained for projected rows and prediction values. This is a reverse label mapping of what is done in Function predict. If the projected row is not present in the sCorrectOut, then all its corresponding input rows should not be selected (Line 23). If a projected row is in output then one or more corresponding input row can be selected. We make the input row with maximum prediction to be selected (Line 19). For each of the other rows, we use the prediction label if it has high confidence (Lines 20-21). Lesser confidence rows are not fed to the decision-tree learner. Finally ID3 is called with the classified input.
The conditions generated using this method are shown below:

- \( (\text{Item} = \text{ItemId}) \) and \( (\text{Year} > 2006) \)
- \( (\text{Item} = \text{ItemId}) \) and \( (\text{Year} > 2008 \text{ or Price} = 7) \)

Note that the first condition is more preferable over the second as it is more compact and not overfitted to a specific value, hence it has more chances of being valid to unseen inputs. In general there can be many solutions given by combinatorial search. As generating all possible repair suggestions and then ranking the solution is time consuming, our approach is to carefully select inputs to the decision-tree learner which can generate good selection condition. The algorithm first ranks the solutions (Figure 9, Line 5) based on the average prediction value of the projected rows that are selected to the \( \text{sCorrectOut} \). In our example, for \( \text{CstId}=2 \), the weight of the first solution, consisting of items \( i_3, i_6 \) is \( (1+2)/2=0.6 \) whereas it is \( (+1+0.1-0.1)/3=0.33 \) for the second solution corresponding to the selection \( i_3, i_4, i_5 \). Based on our heuristics, the first solution is preferred over the second as the first solution has \( i_4 \) which is predicted to be not selected.

The basis of labeling input rows and ranking solutions is same - it is possible to generate better quality condition by following the SVM prediction. See Section 3 for experimental validation of this observation.

The salient features of the final step are summarized below:

- Ranking multiple solutions returned by combinatorial search.
- Assigning the label of input rows based on the labels of projected rows.
- Using ID3 to generate compact selection conditions.

Multiple Occurrences of Faulty Statement  We discuss the variations to the algorithms for the three steps; predict, label, and generate conditions, when there are multiple occurrences of the faulty statement \( s \) in the failure trace, \( \text{Trace} \). Suppose the faulty statement is a READ statement inside the loop in the example shown in Figure 10, which fetches a discount amount corresponding to every row of \( \text{itab} \) and decrements the total order amount accordingly. Suppose there is a fault in the WHERE condition which leads to a wrong report amount for \( \text{CstId}=2 \). There would be as many occurrences of the READ statement in the trace as the number of rows in \( \text{itab} \). Each occurrence has as its input state, a row of \( \text{itab} (\text{wa}) \) and the entire \( \text{DiscTab} \) table (which remains the same for each occurrence). However, the correct WHERE condition

```
14    SELECT CstId Price Year from OrderTab INTO itab
15    SORT itab by CstId
16    DEL from itab where Year = 2009 and Price > 5
17    LOOP AT itab INTO wa
18       AT NEW CstId
19          amount = 0
20       ENDAT
21       READ DiscTab INTO wa_disc WHERE wa.Year = discYear AND discTyp = RETAIL
22       amount = amount + wa.Price
23       amount = amount - wa_disc.discAmt
24       AT END CstId
25       WRITE wa.CstId amount
26       ENDAT
27   ENDLOOP
```

Figure 10: Example for multiple occurrences of faulty statement.
applicable for each occurrence needs be the same. Hence the prediction of SVM could be done in one go with its input as the aggregation of all the rows (the value of wa before each occurrence) and the DiscTab table (refer Figure 11). The joined table, prodTbl, would be itab × DiscTab.

The labeling however would need to be done on a per occurrence basis to generate the an output for every occurrence that finally yields the expected report output. Hence, sCorrectOutSet is of the form \{ < sCorrectOut_{occ1}, sCorrectOut_{occ2}, ..., sCorrectOut_{occN} > \}. The Alloy model would contain unknowns corresponding to each keyCorrectOutSet, which SAT solves for. The decision-tree learning module can also be invoked with an aggregation of the labeled sCorrectOut for each occurrence, since the condition learnt needs to be the same for every occurrence (refer Figure 12).

**Pragmatics** The application of SVMs to the problem at hand requires several steps of data conditioning. The main issue is that SVMs prefer to view data as numerical values for the purpose of distance computation. Relational database tables seldom contain data in this form. We discuss the problems and our solutions.

**Nominal Attributes** The table could contain *nominal* attributes, which are compared for equality, but not for order. For example, a State attribute 2-letter state abbreviation is a nominal attribute. For nominal data, we introduce fresh columns, one for each distinct value of the nominal attribute that appears in tables. For State, we might introduce boolean attributes such as State=AK to State=WY and hide the original State attribute.

At other times, data that looks like non-numeric data might need to be treated numerically. For example, the dates have to be mapped to a numeric interval.

**Key Attributes** Keys are usually nominal data in that value-based proximity of two keys is not meaningful. In joined tables created by Cartesian cross product, one will have two distinct key attributes initially, one coming from each of the two tables. Since it would require too many additional attribute to “de-nominalize” both of these attributes, we instead include an additional boolean attribute that denotes the equality of these two keys (as it is common to have key equality comparison in SELECT statements for a natural join.)

**Scaling** It is typical in use of SVMs to scale data to a normalized [0.0,1.0] range for each attribute. In case the range of data for a certain attribute is very large due a few outliers, care is needed to prevent lower values being scaled down to too close to zero.

**Selecting Relevant Attributes for ID3.** The input tables of the buggy query typically have large number of attributes, many of which are irrelevant. We select a subset of the these attributes that satisfies the following conditions: (1) Contain all the attributes projected by the query (2) Attributes that have been frequently used whenever this table has been used earlier in the code(such as key attributes). (3) Attributes having the same data-type and overlapping values with the state variables at that execution point.

**Seeding Synthetic Attributes in ID3.** Decision-tree based algorithms are only equipped to learn

```plaintext
1. predictMulOcc (sFKeys, sPKeys, sOut, \{ socc1, socc2, ..., soccN \})
2. sIn = sInocc1 U sInocc2 U ... U sInoccN
3. return predict (sFKeys, sPKeys, sIn, sOut, s)
```

Figure 11: Algorithm: Prediction (Multiple occurrences)
clauses that compare attribute values with constants. However, WHERE conditions can contain comparison of two attributes. We seed binary-valued equality predicates between attributes as extra attributes into the learning algorithm. These predicates are seeded based on domain knowledge, for instance, if two tables are being used in a query, a comparison of their key attributes is often a part of the filtering condition. Similarly, attributes of the same data-type and having same range of values may be compared to select records.

3 Evaluation

This section first presents a summary of the experimental results which is subsequently explained using case studies of a select set of subject programs. Finally, it discusses relevant research questions and limitations of our approach.

We selected seven subject programs, which are fragments of ABAP code from industrial applications running on real data sets. The buggy selection conditions, and their fixes are available at Table 4. The bugs in these programs are actual bugs that occurred in the past. In all cases we show the manual fix to the bug.

Our implementation uses the Alloy 4.2 [10] tool-set (specifically, Forge, Kodkod and miniSAT), SVM Light [12] in transductive learning mode, and a home-grown implementation of the ID3 decision-tree learning algorithm. All experiments are conducted in a 2.53Ghz CPU, 4GB RAM laptop running Ubuntu Linux.

The summary of our experimental results is shown in Table 3. For every candidate, we recorded the number of rows in the input table corresponding to the passing and failing keys, shown as #Passing input rows and #Failing input rows respectively in Table 3. Column 4 highlights the prediction accuracy, i.e. the % of failing rows for which the labels were predicted correctly. We also tabulate the total time for correct label computation (dominated by SAT solving times), the combinatorial search space, and the total number of threshold relaxation iterations for all failing keys, and the number of sCorrectOut generated. We note the time to learn a WHERE condition. Finally, we compare the repair suggestion generated by our approach and manual fix to

![Figure 12: Algorithm: Generate Conditions (Multiple occurrences)](image)

---

Table 3: Summary of results. Time in Minutes.
<table>
<thead>
<tr>
<th>Incorrect</th>
<th>Manual Fix</th>
</tr>
</thead>
</table>
| **Ex1** select vbeln posnr aubel aupos matnr netwr  
  from vbrp  
  into table p_i_vbrp  
  for all entries in p_i_vbap  
  where aubel = p_i_vbap-vbeln  
  and aupos = p_i_vbap-posnr. | select vbeln posnr aubel aupos matnr netwr  
  from vbrp  
  into table p_i_vbrp  
  for all entries in p_i_vbap  
  where aubel = p_i_vbap-vbeln  
  and aupos = p_i_vbap-posnr  
  and netwr > 0. |
| **Ex2** delete ADJACENT DUPLICATES FROM db_tab  
  COMPARING kunnr matnr.  
  select from db_tab_rc as db_tab1 ,  
  db_tab_rc as db_tab2  
  where db_tab1 . rc = db_tab2 . rc+1 and  
  db_tab1 . kunnr = db_tab2 . kunnr and  
  db_tab1 . matnr = db_tab2 . matnr  
  and db_tab1 . arktx = db_tab2 . arktx. | delete ADJACENT DUPLICATES FROM db_tab  
  COMPARING kunnr matnr arktx  
  select from db_tab_rc as db_tab1 ,  
  db_tab_rc as db_tab2  
  where db_tab1 . rc = db_tab2 . rc+1 and  
  db_tab1 . kunnr = db_tab2 . kunnr and  
  db_tab1 . matnr = db_tab2 . matnr  
  and db_tab1 . arktx = db_tab2 . arktx. |
| **Ex3** select * from ekbe into table tab_ekbe  
  where ebelp in ebelp_range  
  and zekkn in zekkn_range  
  and gjahr in gjahr_range  
  and belnr in belnr_range  
  and buzei in buzei_range  
  and ( vgabe eq '2' or vgabe eq '3' )  
  order by ebeln ebelp. | select * from ekbe into table tab_ekbe  
  where ebeln in ebelp_range  
  and ebelp in ebelp_range  
  and zekkn in zekkn_range  
  and gjahr in gjahr_range  
  and belnr in belnr_range  
  and buzei in buzei_range  
  and ( vgabe eq '2' or vgabe eq '3' )  
  order by ebeln ebelp. |
| **Ex4** select ebeln ebelp belnr buzei  
  bewtp budat matnr werks ernam  
  from ekbe  
  into table it_ekbe  
  where  
  budat in s_crdate. | select ebeln ebelp belnr buzei  
  bewtp budat matnr werks ernam  
  from ekbe  
  into table it_ekbe  
  where  
  vgabe = 1 and  
  budat in s_crdate. |
| **Ex5** SELECT MATNR ERSDA  
  INTO CORRESPONDING FIELDS OF TABLE IT_MARA  
  FROM MARA WHERE ERSDA > LERSDA and meins = 'EA'. | SELECT MATNR ERSDA  
  INTO CORRESPONDING FIELDS OF TABLE IT_MARA  
  FROM MARA WHERE ERSDA > LERSDA and meins = 'EA'  
  and ERSDA < HERSDA. |
| **Ex6** select vbeln posnr matnr netwr waerk  
  from vbap  
  into table p_i_vbap  
  for all entries in p_i_vbak  
  where vbeln = p_i_vbak-vbeln. | select vbeln posnr matnr netwr waerk  
  from vbap  
  into table p_i_vbap  
  for all entries in p_i_vbak  
  where vbeln = p_i_vbak-vbeln.  
  and werks = p_p_werks. |
| **Ex7** SELECT * FROM ekbe INTO TABLE tab_ekbe  
  WHERE ebeln IN ebelp_range  
  AND ebelp IN ebelp_range  
  AND zekkn IN zekkn_range  
  AND gjahr IN gjahr_range  
  AND belnr IN belnr_range  
  AND buzei IN buzei_range  
  AND ( vgabe EQ '2' )  
  ORDER BY ebeln ebelp. | SELECT * FROM ekbe INTO TABLE tab_ekbe  
  WHERE ebeln IN ebelp_range  
  AND ebelp IN ebelp_range  
  AND zekkn IN zekkn_range  
  AND gjahr IN gjahr_range  
  AND belnr IN belnr_range  
  AND buzei IN buzei_range  
  AND ( vgabe EQ '2' OR vgabe EQ '3' )  
  ORDER BY ebeln ebelp. |

Table 4: Real-world examples of errors in selection statements.
determine whether our suggestion is useful to arrive at the manual fix.

3.1 Case Studies

We describe details of applying our approach to four subjects to highlight some of its key characteristics and how it handles different types of faults.

Scenario 1: Accurate predictions. We start with a simple case where a straightforward application of our approach produces a repair very quickly.

Consider the following buggy SELECT statement in Ex3,

```sql
select * from ekbe into table tab,ekbe
    where ( vgabe eq '2' or vgabe eq '3' )
    //and ebeln in ebeln_range. Needed in the correct query
```

The WHERE clause is essentially missing two predicates in the form of a missing range-check predicate for the field `ebeln`. This error results in 12 unexpected rows in the output of the program.

Correct Label Computation. The incorrect rows in the program output correspond to 12 failing rows in the input `ekbe` table. Each row corresponds to a unique key value, so there are 12 `sFkeys`. The number of passing key rows are much higher (316 `sPkeys`). The predictions were 100% accurate. All the 12 `sFkey` input rows were given negative prediction values by SVM. Our algorithm assigns all the rows with negative labels in the first iteration. This resulted in the correct output within a minute, i.e. labeling was found satisfiable by SAT.

Decision-tree learning. The decision-tree learning module was invoked with the passing rows labeled as per the existing output of the statement and the failing rows marked with negative labels. The condition learned was as given below and was correct in not selecting exactly the 12 failing rows.

```sql
vgabe = '2' and
ebeln <= 450000229
```

The generation of a comparison on `ebeln` conveys to the programmer that a bound check is missing. Indeed, the source code defines the constant `ebeln_range` as `[ebeln_low, ebeln_high]`, but the buggy query fails to use it. Because of the data-distribution in this test case, the generated repair condition does not contain the the lower bound on `ebeln`, nor could it generate the additional condition on `vgabe`. However, the generated condition presents an useful hint to the user to add the missing bound check on `ebeln`.

Scenario 2. SELECT with table joins. This scenario illustrates a case where highly accurate prediction reduces the combinatorial search space when the buggy statement involves table joins.

The program (Ex1) creates a sales order report by calculating order amount and unbilled amount for each sales order. It first creates a table called `p_i_vbrp` using the following query:

```sql
select vbeln posnr aubel aupos matnr netwr
from vbrp, p_i_vbap
into table p_i_vbrp
where aubel = p_i_vbap-vbeln
and aupos = p_i_vbap-posnr
    // and netwr > 0. Needed in the correct query
```

However, the missing `netwr > 0` predicate from the WHERE condition causes incorrect `p_i_vbrp` formation, shown below for the key `aubel=102`.
The computation after the SELECT reads the rows \(<102, 20, 0>\) and \(<102, 30, 0>\) corresponding to two posnr values 20 and 30, instead of \(<102, 20, 8000>\) and \(<102, 30, 11200>\), which it would have read from the correct version of the table. This leads to the incorrect output for the failing key 102. Altogether there are 18 failing keys in this example.

**Correct Label Computation.** There were 40129 passing input rows that were labeled as per their outcome in the existing execution. 10032 failing input rows were unlabeled. SVM attached a positive prediction to 19 unlabeled rows and a negative prediction to the remaining. As noted in Table 3, in this case the prediction is highly accurate (99.9%), leaving only 3 input rows incorrectly predicted. Each of these rows corresponds to a failing key and the maximum predicted row in the block for a projected row. Thus for each of the 3 failing keys, a projected row was incorrectly labeled. For these three keys, the iterative threshold adjustment process was invoked until a correct solution was found. For example, for key 146, initially both rows were marked negative (as shown below) leading to unsatisfiability.

<table>
<thead>
<tr>
<th>aubel</th>
<th>aupos</th>
<th>netwr</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>146</td>
<td>10</td>
<td>0</td>
<td>0.99972347</td>
</tr>
<tr>
<td>146</td>
<td>10</td>
<td>30</td>
<td>0.39996994</td>
</tr>
</tbody>
</table>

With a threshold of 0.4, the second row with confidence less than 0.4 was assigned an unknown label (to be determined by SAT), while the first row was given negative label. The same was done with the other two failing key records. SAT assigned positive labels to these rows leading to satisfiable solutions. In all, this process completed in 4 minutes. The total search space for the SAT solver is \(3^2 \times 1 = 6\).

Here we also give an estimate of the combinatorial search space if predictions had not been used at all. For each of the 18 failing keys, on average there are 3 projected rows, where each projected row maps to a block size of 227 input rows. For each failing key, the state space for choosing the set of projected rows that yield the correct output is \(2^3\). Each solution comprising of projected rows, needs to be mapped to input rows of the joined table, before being fed to the ID3. The state space for this would be \(227^3\) in worst case, when the solution contains all three projected rows. Thus the total search space, in worst case, to generate correct condition without using any predictions would \(18(2^3 + 227^3)\). As explained earlier, use of predictions helps reduce this space to just 6.

**Decision-tree Learning.** Decision-tree learning discovered the correct WHERE condition:

aubel = p_i_vbap-vbeln and aupos = p_i_vbap-posnr and netwr > 6

Note that our approach was able to learn the join condition. The row selected from each block corresponding to a unique projection value, is critical in determining the correct join condition. We would like to highlight that for the given data, the row with maximum prediction value corresponding to each block, was the only one which satisfied the join condition. Hence the selection of any other row from the block would have not lead to the discovery of the join condition. This adds evidence to the fact that our design decision of selecting the row with maximum prediction value, would result in producing high quality conditions close to the ideal.

The constant discovered is 6 rather than 0, due to the distribution of the data. But it is a good repair suggestion since it points out an important missing clause.
Scenario 3. Use of predictions to rank candidate solutions. This scenario highlights that our repair algorithm is not restricted only to SELECT statements, and further illustrates a case where predictions aid in reducing the space of candidate solutions on which decision-tree learning has to be performed.

The buggy statement in this example (Ex2) is a DELETE statement, shown below.

```
DELETE ADJACENT DUPLICATES FROM db_tab
//arktx Needed in the correct query
```

The DELETE ADJACENT DUPLICATES statement deletes a row from the table that has same values in its immediately previous row for the fields specified in COMPARING clause. This could be modeled as an equivalent SELECT statement as shown below.

```
select * from db_tab_rc as db_tab1, db_tab_rc as db_tab2
where db_tab1.rc = db_tab2.rc+1 and
  db_tab1.kunnr = db_tab2.kunnr and
  db_tab1.matnr = db_tab2.matnr and
// db_tab1.arktx = db_tab2.arktx Needed in correct query
```

Where db_tab_rc has an extra column rc in addition to all the columns of db_tab. It contains the same records as db_tab with the rc column populated with the row number.

This statement selects rows that would need to be deleted by the original statement. The code after the DELETE statement, in a nutshell, aggregates the netwr amounts corresponding to every unique value in monat field on db_tab. The output report had incorrect amounts displayed for two monat values - Sep2008 and Oct2008 (2 failing keys).

Correct Label Computation. The db_tab had 10 records with monat as Sep2008 and 20 records with Oct2008. Note that although the SELECT is over a join and every row of db_tab_rc maps to a block of rows in the joined table, we know upfront the exact record that needs to be considered from every block. The only record that can be selected in the block corresponding to every failing row of db_tab is the one where db_tab1.rc = db_tab2.rc+1 is satisfied as this predicate will be present in the correct version of the query. Hence the search space remains 10 and 20 respectively for the two failing keys.

Label Predictions. There are only 80 positively labeled passing key records compare to 12691 negatively labeled records in the joined table. Hence the accuracy of predicting positive labels was low. All rows were assigned prediction values -1.0 and -1.0000001. The first iteration assigning all failing key rows with negative labels was unsatisfiable. However, with a threshold of 1.0, rows with prediction values -1.0000001 were marked negative, while the remaining were assigned labels based on SAT-based search. For Sep2008, there was exactly one correct solution, while for Oct2008, there were 32 possible correct solutions. The reason for the large space of possibilities in the latter case was that there were 5 records with netwr amounts of 0. The presence or absence of each of these records yields the same final output. Hence SAT produces 32 different solutions yielding the expected output.

It would be inefficient to generate 32 possible WHERE clauses. This is where predictions aid in heuristically selecting the solution that is most likely to yield the ideal WHERE clause.

Label Predictions-based solution ranking. The 32 solutions were ranked based on their average prediction values. The solution with the highest average prediction value is the one whose constituent rows have the highest likelihood of being assigned positive labels. In practice, the decision-tree learning would be invoked on the highest ranked solution first, the condition presented to the
user and conditions for the remaining solutions would be generated only if the user is not satisfied with the first condition. However, for our experiment we invoked the decision-tree learning on all the solutions to evaluate the efficacy of our selection heuristic.

**Decision-tree Learning.** The WHERE clause of the SELECT statement, learnt for the highest ranked solution was,

\[
\text{db_tab1.rc = db_tab2.rc + 1 and db_tab1.kunnr = db_tab2.kunnr and db_tab1.arktx = db_tab2.arktx}
\]

As can be observed, the condition on `arktx` that was missing in the incorrect version is correctly discovered. However, the condition on `matnr` is missing from the learned clause. This is because the `matnr` values are equal for all adjacent records in which the other two conditions are also satisfied. This makes the learned WHERE clause correct for the given input set. The conditions learnt for the other solutions with lower ranks were of poorer quality as can be observed by their sizes, shown left.

**Scenario 4. Impact of incorrect selection on passing keys.** `Ex4` displays an interesting scenario which violates our assumption about the correctness of erroneous SELECT statement for the passing keys. The final output corresponding to passing keys is still correct but the SELECT acts incorrectly on some of them.

The erroneous SELECT statement given below leads to the inclusion of 58 extra records for the failing keys in the actual output of the program, compared to the expected correct output.

```sql
select ebeln ebelp belnr buzei bewtp
budat matnr werks ernam
from ekbe into table it
where budat in s_crdate
// AND vgabe = 1 Needed in the correct query
```

In this example, for the passing keys too, the erroneous SELECT statement selected some extra rows, but subsequently they got deleted by a DELETE statement in the program. Consequently, these passing keys yielded the correct final output anyway.

**Correct Label Computation.** The incorrect labeling for 16 passing key records where `vgabe = 1` impacts the accuracy of predictions as seen from Table 3. Hence the approach of using predictions to label records performs poorly. The algorithm passes through 4 iterations of threshold adjustment and produces correct solution only when all records are labeled based on SAT-based search.

**Decision-tree Learning.** The incorrect labeling of the passing key records impacts the WHERE clause condition learned.

The condition is quite different from the one in the correct version of the code. It leads to the expected final output on this data set, but this is not a useful repair suggestion.

### 3.2 Discussion

Based on our experimental evaluation we address the following key re-
**RQ1. Do the predictions based on the data distribution aid in finding the correct output for the failing keys efficiently?**

In all cases, prediction based labeling of rows helps in determining a correct output state to the faulty statement within 8 minutes in worst case.

The efficiency of our algorithm is attributed to high prediction accuracy which effectively reduces the combinatorial search space, and further design decisions such as: 1) an iterative threshold relaxation strategy which judiciously un-labels incorrect predictions. The low number of iterations (in most cases) suggest that there were only few incorrect predictions that needed to be labeled by SAT, 2) ranking of solutions based on predictions which saves the effort of generating conditions corresponding to all the solutions.

As noted in Scenario 2 (Ex1), in the absence of predictions, in the worst case, a combinatorial search based strategy has to explore huge search space to arrive at useful solution. Even for subjects that do not involve joins predictions-based labeling brings about significant reduction in search spaces: 87% reduction from a total of $2^{10} + 2^{20}$ for Ex2, and almost 100% reduction from a total of $2^{84}$ for Ex5.

**RQ2. How useful are the repair suggestions?**

The usefulness of the generated repair suggestions is summarized in the last column of Table 3. Except for Ex4, the repair suggestions were close to manual (ideal) fixes for the bugs. We have already discussed the 4 cases (Ex1, Ex2, Ex3, Ex4) in our case studies. For Ex6, we show below the manual fix, the condition generated by our approach (with predictions). In this case p_p_werks is a parameter to the program which had a value GBS1. As can be seen, the condition generated based on predictions is very close the ideal fix.

<table>
<thead>
<tr>
<th>Ideal fix for Ex6:</th>
<th>Condition learned from a solution based on predictions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vbeln = p_i_vbak-vbeln$ and $werks = p_p_werks$</td>
<td>$vbeln = p_i_vbak-vbeln$ and $werks = \text{`GBS1'}$</td>
</tr>
</tbody>
</table>

Below we show few conditions generated using combinatorial search without prediction for Ex6. Clearly such solutions are far away from the ideal fix.

<table>
<thead>
<tr>
<th>Conditions learned from other solutions</th>
<th>Conditions learned from other solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vbeln = p_i_vbak-vbeln$ and (werks = \text{<code>GBS1'}$ and $(vbeln &lt;= 102.0$ and $(\text{waer = </code>EUR'}$ or $(\text{waer = `USD'}$ and $(\text{posnr &lt;= 15.0$))) or $(vbeln &gt; 102.0$))</td>
<td>$vbeln = p_i_vbak-vbeln$ and (werks = \text{`GBS1'}$ and $(vbeln &lt;= 102.0$ and $(\text{netwr &lt;= 3524.4}$ or $(\text{netwr &gt; 6336.4})$ or $(vbeln &gt; 102.0$))</td>
</tr>
</tbody>
</table>

The reason for high quality of our repair suggestions can be attributed to the labeling of failing-key data based on its proximity to passing-key data which generates the conditions that classify regions of data uniformly, which is typical of WHERE clauses. Even though theoretically it is possible to generate many $\text{sCorrectOuts}$ which generate the expected correct output of the program, our approach only generates few of them based on the prediction. This in turn generates few good conditions which are close to the ideal fix.
RQ3. Is syntactic mutation technique feasible for real data?

To check the feasibility of mutation-based repair, we consider a repair strategy which checks if the WHERE condition could be corrected by either adding one clause, removing one existing clause, or replacing an existing clause with a new one. Clauses of the form Field Operator Field, Field Operator Constant and Field Operator Variable are considered as mutants.

In almost all the cases the search space of the number of mutants is very huge (in the order of 40,000) leading to a blow-up in the worst-case exploration time (in the order of 40 hours assuming an average of 5 seconds to execute 1 mutant). The main reason being the large number of distinct values that could be compared in the clauses of the form Field Operator Constant. An algorithm that does not consider clauses that involve constants would work much faster, however it would be unsuccessful in discovering the correct WHERE condition for Ex1, Ex3, and Ex7.

Limitations

1. Our technique assumes that the incorrect selection criteria works correctly for keys that satisfy the final output correctness criteria. Violation of this assumption (Ex4) impacts the quality of the predictions and the WHERE condition learned.

2. Sufficient amount of diverse passing data is required to make the learning effective. For example, in Ex7, the failing rows were all erroneously predicted to be negatively labeled. This is because majority of the rows for the passing keys were negatively labeled; very few rows were positively labeled.

3. Attention must be paid to data conditioning, which currently uses heuristics based tuning (described in Pragmatics in Section 2) to arrive at good label prediction.

4. ID3 algorithm is designed to correctly label all training data. However, if the data in the current execution is not representative enough, then the WHERE condition created may be overfitted to the data (Ex4). Techniques to avoid overfitting [20] compromise the accurate labeling of training data. Finding the right balance for our application is the subject of future work.

5. Our algorithm strives to generate the most compact classifier for the given data. In some cases, this could exclude clauses that would be in general necessary, but do not impact the outcome for the given data. To reiterate, our technique generates useful repair suggestions and not necessarily plug-and-play repairs.

4 Related Work

Recent years have seen much progress in techniques for automated debugging – both for fault localization [15], i.e., finding the locations of (likely) faulty lines of code, as well as program
repair [25], i.e., correcting the faulty lines of code to fix the fault(s), which is the focus of this paper.

**Fault localization.** The application of machine learning to debugging is largely confined to fault localization. Decision tree generation algorithms, including C4.5, have been used in conjunction with the fault localization tool Tarantula [14] to cluster failing tests in order to help developers manually fix bugs in their code more effectively [14, 3]. Statistical debugging techniques [18, 11] employ statistical analysis on the data collected from passing and failing program runs to determine likely faulty statements.

**Program repair.** The problem of program repair has been the focus of a number of recent techniques, including those based on evolutionary algorithms [25], specifications [6], program code transformations [5], as well as program state mutations [4]. The key novelty of our technique with respect to previous work is two-fold: (1) previous work has not considered repair of SQL statements, in general, and ABAP programs, in particular; and (2) machine learning and systematic search have not been integrated before for program repair.

Weimer et al. [25] introduced the idea of program repair using genetic programming, where existing parts of code are used to patch faults in other parts of code and patching is restricted to those parts that are relevant to the fault. Ackling et al. [1] repair a program by evolving patches to fix it rather than evolving the faulty program itself, and argue that doing so simplifies the repair problem. Wilkerson et al. [26] present a co-evolutionary approach where code and its tests are co-evolved to improve the bug finding ability of tests as well as to improve the overall quality of the code in order to provide an automated software correction system.

Chandra et al. [4] use changes to program states in a faulty program to approximate the behavior of a correct program with respect to a given set of passing and failing tests, and use these state mutations to guide syntactic changes to code in order to repair it. Malik et al. [19] use a search-based technique for data structure repair [16] as a basis of program repair. Specifically, they use mutations done on program state to fix corrupt data structures as a basis of synthesizing program statements that abstract those fixes using program variables. Gopinath et al. [6] consider a similar setting of repairing programs that operate on structurally complex data but use a different approach. They introduce nondeterminism in the program’s operations and use SAT solvers to generate valid program states (with respect to given specifications), which are then abstracted into program expressions that evaluate to those states and provide the fixes. Jobstmann et al. [13] originally used this technique to replace faulty program expressions with unknowns and formed a model checking problem in order to repair a faulty program with respect to its linear time logic specification. Griesmayer et al. [7] map the problem or repairing boolean programs to finding a memoryless, stackless strategy in a game and explore the game graph to find a repair for the boolean program, and show how it can be used to repair a class of C programs.

Debroy et al. [5] introduced the idea of using mutations, i.e., syntactic transformations to the faulty program as a basis of repair. The developed their technique in the context of the Tarantula tool and spectrum-based fault localization using a given set of passing and failing tests to focus mutations. While such code transformations can assist in debugging, the space of variations to explore grows very quickly the feasibility of using such a technique for real applications requires developing novel pruning techniques.

Wei et al. [24] attempt to combine specification-based and test-based repair. Boolean queries
are used to build an abstraction of the state, which forms the basis to represent contracts of the class, fault profile of failing tests and a behavioral model based on passing tests. A comparison between failing and passing profiles is performed for fault localization and a subsequent program synthesis effort generates the repaired statements. This technique however only corrects violations of simple assertions, which can be formulated using boolean methods already present in the class.

**Program synthesis.** A closely related area to program repair is *program synthesis* [9], where a goal is to generate (parts of) a program independently of a given incorrect version. A number of program synthesis techniques are based on specifications. Programming by sketching [22] employs SAT solvers to generate missing parts of a given skeletal program with respect to another reference program that serves as a specification. A SAT solver completes the implementation details by generating expressions to fill the “holes” of the skeletal program by exploring several of its variants. Gulwani et al. [9] use the counterexample guided iterative synthesis paradigm together with SMT solvers to synthesize loop-free programs with respect to given specifications of desired functionality. Kuncak et al. [17] generalize decision procedures into synthesis procedures to synthesize code snippets from specifications.

To alleviate the burden of writing detailed specifications, some recent techniques support synthesis based on given concrete input/output examples. Gulwani [8] presents such a technique for synthesizing string processing code for spreadsheets using examples of how a user processes sample strings. More recently, Singh et al. [21] integrate *scenarios*, which illustrate steps of modifying specific data structure instances, with given code skeletons and inductive definitions to facilitate program synthesis.

At present, techniques for synthesis have largely been developed independently of techniques for repair.

### 5 Conclusion and Future Work

We presented a novel approach to generate repair suggestions for defective database programs, where the faults are in the selection condition of database statements. We use techniques from machine learning to learn a decision tree from the correct behavior shown on the defect-free data as well on correct behavior determined for defect-inducing data. The decision tree guides our repair technique. Our key novelty is to determine the correct behavior of the defect-inducing data using a combination of SAT-based search and prediction generated by support vector machines (SVMs). Experiments using a prototype embodiment of our approach on a suite of real programs show the promise it holds in automated debugging.

While our current work focuses on repairing database statements, we envisage an extension of our idea of data-driven repair to more general imperative programs. A direct application would be to correct faulty branch conditions that are covered by both passing and failing tests. The outcomes of the condition on faulty runs could be predicted based on passing runs, which in turn could be used to learn a condition that works correctly for all the tests.
References


6 Appendix
ABAP code semantics:
The code after faulty statement s until the Exit of the program can be represented as
\{Stmt1; Stmt2; ...; Exit\}

Stmt1 := Imperative Logic Statement (StmtIMP) or DB statement (StmtDB)

StmtIMP :=
IMF-form1 := Var = Exp, where Exp = Constant/Var'/Exp arithmetic operator Exp
IMF-form2 := IF THEN ELSE ENDIF
IMF-form3 := LOOP AT Table-var INTO WA-var [AT NEW/ON CHANGE {Stmt1} ENDAT {Stmt1}]
IMF-form4 := PERFORM <sub-routine> using <variable names>

StmtDB :=
DB-form1 := SELECT Projection(Table1) FROM Table1 INTO Table-var [FOR ALL ENTRIES IN TABLE2]
WHERE condition
DB-form2 := DELETE ADJACENT DUPLICATES FROM Table1 COMPARING Table-field1[.Table-field2 ...]
DB-form3 := READ TABLE Table1 INTO wa with Table1_key = Var

Model State as Relations:
1. Model non-table local variables as singleton scalars (relations with arity=1, size =1).
2. Model tables as sets of binary relations. \{<Table1_key, Table1_field1, ..., Table1_field_n>\}
   modeled as \{<Table1_key, Rel1_field1, ..., Rel1_field_n>\},
   - Rel1_field := Row-index \rightarrow set_field,
   - Row-index \subset Integer,
   - set_field \subset Integer if fieldi is a number or
   - set_field := set of all elements of fieldi ( {Name0,Name1, ...,Name} for Name field).
3. sCorrectOut is modeled as \{ri\}. i = 1 to R (max # of rows in sIn).
   ri = \{\} if labeli = false,
   ri = Row-index; if i = true

Model Computation as Relational operations:
1. Inline subroutines. Unroll loops specified # times to create a directed acyclic graph of the control flow.
2. Model each Stmt in each path as a declarative constraint relating input and output
   states of the statement.

Relations_s^in = Constraint(Stmt_s, Relations_s^in)

3. Constraint(StmtIMP) =
i. IMF-form1:
   Rel_var = \{<Constant>\} for Var = Constant, Rel_var = Rel_var, for Var = Var'.
   int(Rel_var) arithmetic operator int(Rel_var) for Var = Var arithmetic operator Var2.
   Rel_var = int(Rel_var) operator Constant for Var = Var operator Constant
ii. IMF-form2:
    (Cond \Rightarrow Constraint(Stmt1))V(~Cond \Rightarrow Constraint(Stmt2))
iii. IMF-form3:
    for every rownum in \{Row-index\},
    Value of fieldi = rownum . (Row-index \rightarrow set_field).
    (AT NEW, ON CHANGE are handled comparing the values between successive rows to
    identify change.)

4. Constraint(StmtDB) =
i. DB-form1, DB-form3:
   all r s.t. (r in Table-var) \iff (r in Table1[projection(cols)]) \land (cond = true)),
   all r s.t. (r in Table-var) \iff (r in Table1[\rightarrow Table2]) \land (cond = true))
ii. DB-form2:
   Table1 modeled as TableName1, TableNameNew2 = Table1 with an added field called rowNum
   all r s.t. (~r in TableNameOut) \iff (r in TableNameNew1 \rightarrow TableNameNew2)
   \land ((cond = true) \land (TableName1.rowNum = TableNameNew2.rowNum))

5. Constraint(path_n) = Constraint(Stmt1) ... Constraint(Stmt_m),
   Constraint(P) = Constraint(path_1) V ... V Constraint(path_n)
   return Constraint(P)

Figure 13: Algorithm: ABAP to Alloy Model