ABSTRACT
Diagnosing faults in embedded queries in database applications is a daunting process. When test cases fail, the traditional way of diagnosing faults is to follow possible execution paths, either mentally or step-by-step in a debugger, to locate the problematic area. The diagnosis problem becomes even harder when you have embedded language with quite different semantics and properties.

Our focus is on a specific problem: diagnosing failed test cases caused by embedded queries in database applications which are syntactically correct but semantically incorrect (i.e., they produce incomplete or incorrect results). Much research literature is available on database applications and databases but the diagnosis problem for embedded queries that cause failure of test cases has not been tackled.

We perform an experiment to see how far existing techniques could be useful in proposing a new technique for this problem. We identify the additional components that need to be developed to take us to a full solution and describe our tentative conclusions so far.

Categories and Subject Descriptors
d.2.8 [Software Engineering]: Testing and Debugging, Embedded Queries—Faults

General Terms
Experiments

Keywords
Testing Database Applications, Testing, Diagnosing Faults

1. INTRODUCTION
The aim of our research is to diagnose faults in queries embedded in application programs and propose changes to such queries which when applied can make the test cases pass. We elaborate on this problem as below.

Database applications interact with databases using SQL statements embedded in the code. Such SQL statements impose additional difficulties for testing as compared to testing of general applications. The problem we specifically focus on is diagnosing failed test cases that occur when embedded queries are syntactically correct but semantically incorrect (i.e., they produce incomplete or incorrect results). Even unit testing tools specifically designed for use in testing database code (e.g., DBUnit1, SQLUnit2) do not make use of SQL semantics to go beyond giving an indication of where the error which has occurred is in the code. While such tools indicate where the error becomes visible, they do not tell us what the fault revealed actually is. The first question is, how do errors in queries manifest themselves in test results? The typical answer is: as an unexpected value being returned from one or more methods, which triggers failed assertions in one or more test cases. The next question is: why did the methods return the unexpected values? The answer could be because of a syntactically correct but semantically incorrect query returning wrong results. In such cases the developers have to check the failed assertions and the methods called by the assertions, which interrogate the database to determine the result values. The traditional way of diagnosing faults is to follow possible execution paths, either mentally or step-by-step in a debugger, to locate the problematic area. Diagnosing faults in the source code manually can be challenging and time consuming for many developers. We need to develop automated techniques for diagnosis of such faults. Once we have such techniques then the next task for us would be to find out how changes to faulty queries can be proposed which when applied can make the failing test cases pass. Automating this whole process should save developers' time.

The rest of the paper is arranged as follows. Our breakdown of the problem is described in section 2. In section 3 we describe related research. We explain our experimental results so far in section 4, and our solution approach is outlined in section 5. We conclude our paper in section 6.

2. RESEARCH PROBLEM
In this section, we give a breakdown of our research problem and identify our key research questions. We also explain the tasks which we need to undertake for finding the answers for these questions. First we define the notions of ‘fault’ and ‘fault in embedded queries’.

Fault: A program P is faulty if output from the program

1http://www.dbunit.org
2http://sqlunit.sourceforge.net/
against some input is different from the expected output.

Fault in Embedded Query: An embedded query is faulty if faults are manifest in the program that result from a fault in the query.

Our research problem can be stated as follows.

Given:
1. the source code of a database application program, including the schemas and other metadata for any databases it accesses, and
2. the source code of a test suite for the application program, and
3. a set of test results from executing the test suite, showing which tests passed and which failed,

Can we:
(a) determine which (if any) embedded queries are responsible for the failed test cases?
(b) compute the set of changes to the responsible queries that cause all test cases in the suite to pass?

To further illustrate this problem, Figure 1 shows a component (DFQ - Diagnosis for Faults in Queries) that implements a solution to the problem as a black-box. It shows the inputs and outputs of the process. We execute the test cases on the application source code and some of the test cases fail. We record the test results and need to find query statements responsible for the failed test cases. We have to identify the nature of the faults (such as incomplete, incorrect or missing tuples) in the responsible queries by identifying which tuple is implicated or which attributes are involved. We need to compare such attributes with the attributes in the original query to see what changes we would need (refined queries), based on which corrections may be applied to make the test cases pass.

![Figure 1: Research problem.](image)

In next section we identify the research questions and elaborate our research problem further by looking inside DFQ box.

### 2.1 Research Questions

To solve our problem, we must answer the following research questions (RQs):

**RQ 1:** How can we identify sets of embedded queries in program P that are candidates for the location of the fault in the queries?

As program P consists of number of different statements written in imperative languages like Java with queries embedded, and we need to find the suspicious query statements (i.e., those that may contain the fault), so we have to separate the query statements from the non-query statements to obtain a set of candidate faulty queries.

For each candidate faulty query:

**RQ 2:** How can we form a hypothesis as to the nature of the fault in the query, that is supported by the specific details of the failed test?

After identifying candidate faulty queries, we need to find the nature of the fault in the queries. The term ‘nature of the fault’ is to determine if the fault is caused due to incorrect, incomplete or missing tuples?

**RQ 3:** How can we use this hypothesis to propose refined forms of the candidate faulty query that may cause all tests in test suite to pass?

If we know the nature of the fault in the candidate queries then we can identify the potential changes to such queries basing on the information of the failed test cases and the original queries. Such changes would require verification & validation (V&V) to see if those potential changes make the failing test cases pass or not. The next question is:

**RQ 3.1:** How many ways are there to refine the candidate faulty queries in the P that might correct the hypothesized fault?

There are many ways in which a query can, potentially, be faulty e.g., returning incorrect tuples or failing to return some significant tuples. We need to explore if static analysis only can find a set of refined queries or any existing techniques/tools or combination of both would be required. Once we find the way to obtain set of refined queries then the next question is:

**RQ 4:** How can we determine whether the candidate refinements proposed for the query will indeed cause all tests in the test suite to pass?

The set of refined queries found in answers to RQ 3.1 would need V&V to see if some of them fulfill the requirements of making the failed test cases pass and do not affect the results of the remaining test cases. Our final question is:

**RQ 5:** How can we best convey to the user the information about the candidate refinements required to make all the failing test cases pass?

Once the candidate refinements are V&V then we need to find a mechanism to inform the user about the faulty queries and recommend alternative queries/changes to be applied to make the test cases pass. So he/she can decide which to apply.

![Figure 2: Breakdown of our research problem.](image)

The tasks implied by these questions are further illustrated...
in Figure 2. The rectangular boxes represent the components which we need to build based on the answers to our research questions. The figure contains three parts which are: rank queries by suspiciousness, find refined queries and apply validated refinements.

When the test cases are executed and some of them fail, we need to identify suspicious (or faulty) statements, and from these statements we need to extract the candidate faulty queries to answer RQ 1 (by filtering out the non-query statements). The next task is to analyse the candidate queries to identify the types of faults (hypothesized faults) in those queries to answer RQ 2. These faults/culprit tuples can be given to the “Refine queries” component, to be used to obtain a set of refined queries to answer RQ 3 and 3.1, which can be presented to the user by our component “Present query information to user” after validation by component “Check queries pass tests”. The types of information which can be presented to the user are: information about the faulty queries/test cases, recommended alternative queries or prospective changes required to the code and the test cases which would pass because of such changes. Upon selection by the user the changes can be applied to the source code by component “Apply changes” and the test suite re-run to ensure everything is applied correctly.

In summary we have identified our research questions for which we need to find answers to solve our problem. We have also identified potential components/techniques which we need for our overall solution.

3. RELATED WORK

In this section, we explore existing techniques to see how far these can be helpful in solving our problem, thus, discovering, which components we would need to develop ourselves.

Testing of database applications is different from general software testing. The main difference is in the test environment that includes a database which brings additional challenges due to persistent nature of database state. There has been a considerable amount of work on testing database applications and databases. The areas covered include test case generation [7, 23], test database generation [7, 4], test dataset generation [21] and efficient test execution [11, 20, 8, 10]. In recent years, tools/techniques like AGENDA [7], SUITE [2], WHODATE [17], SQLFpc [5], SQL Coverage Measurement [19], BoNuS [15], HTDGen/HTTrace/HTPar [3] have been developed that cover topics in the above areas. These systems do not tackle the diagnosis problem for embedded queries (or for other kinds of interaction with database). However, proposals exist in the literature which are helpful in covering parts of the full embedded query diagnosis problem we outlined in the previous section. In particular, work in two other areas provides some help with our problem:

- Work in defect localisation for general software may be able to help us locate candidate faulty queries.
- Work in query refinement/provenance may help us propose alternative query forms that correct problems revealed by failed test cases.

In this section, we broadly review work in these two areas, and indicate how far they can solve our problem.

3.1 Defect Localisation

There are no proposals in the current literature for determining specific queries that are responsible for failed test cases, but a more general class of tool exists which can determine the general program statements that may be responsible. This task is called “defect localisation”, and several proposals for how to achieve it have appeared in the literature.

The existing techniques find a subset of statements which are likely to be faulty. One such previous work, called Set Union and Set Intersection, by Agrawal et al., computes the set difference of the statements covered by passing and failing test cases [12]. The result is an initial set of suspicious statements which can be searched for faults. There is also related work by Renieris and Reiss [18], which is similar to Agrawal et al.’s. However, their technique compares the statements executed by a failed test case with those executed by the passing test case that has coverage most similar to the coverage of the failed test case. The set of statements executed by the passing test case is removed from the set of statements executed by the failed test case, leaving those to be used as an initial set of statements for searching for the fault. Cleve and Zeller’s technique, called “cause transition”, focuses on the differences between the program states of passing and failing runs of the program. This technique finds variables, and values stored in the variables, at points of the execution of a program that are relevant to the failure, and the moment in time where cause transition occurs [6].

The cause transition is the set of program statements that causes the transition, a change in behaviour of the program, followed by the failure; such points in a program are then used as the initial set of points to start to search for faults.

All of the above techniques pertain to general software applications and are used to find the initial points in a program to start the search for suspicious statements causing the failures. They may be helpful as a partial solution for the beginning part of our problem. However, we plan to use a more recent technique, called “Tarantula”, which has features that suit our problem better. Tarantula uses test cases, test results, information about all the statements executed by each test case, the source code of the program under test and calculates the suspiciousness of each statement by utilizing the percentage of test cases that execute a statement [14, 16]. It ranks the statements by assigning each one a score between 0 and 1, where 0 is the most suspicious and 1 is the least suspicious. A key difference about Tarantula is that it does not identify a subset of suspicious statements, but instead scores each statement. It tells the programmer what is a good order in which to examine the statements. Because of this, the defect will always be found, which it would not be if the programmer is only given a subset of statement to examine, which may not contain the faulty statements. The more suspicious statements should be considered first by the programmer when looking for the fault.

Tarantula is helpful in identifying the suspicious statements but we need to find suspicious embedded queries in a program. Once we identify the suspicious queries, the next task is to find out if the query is returning the desired results, incomplete tuples or incorrect tuples or has missing tuples.

3.2 Query-Based Provenance

In order to find solution for expected/unexpected query results we need to explore work in data provenance, which attempts to explain why a tuple appears in a query result or why it is not.

There are two main models for explaining why-not questions. The first is to identify the query operator that is responsible
for eliminating the missing tuples from the results [1]. The second model explains a missing tuple in terms of modifications to the database which causes the tuple to appear in the query result [13]. A technique which combines both of the above models is that by Tran and Chan [22]. This technique (illustrated in Figure 3) takes the query, the database schema and user options/missing tuples as input and automatically generates a set of refined queries whose results include both the original query’s result and user-specified missing tuples.

![Query Diagram](image)

**Figure 3: Constraint-based Query Refinement - ConQueR.**

This technique only handles the missing tuples problem, so we need to find techniques for the other faults, such as incorrect tuples or incomplete results. Therefore, we need to develop another component as shown in Figure 2, which can provide refined queries for those faults which are not handled by ConQueR.

4. FAULT LOCALISATION FOR EMBEDDED QUERIES

In this section, we describe the aims of the experiment with Tarantula in terms of the identified research questions. Then, the obtained results are reported and analysed.

4.1 Experiment Description

We performed an experiment with Tarantula to see how far it could solve the first part of our problem. We set out to determine if Tarantula can localise faults in embedded queries (addressing RQ1).

The experiment was performed using a simple application for holiday tour booking, containing embedded queries in the application code. This sample Java system consists of 14 database tables, 67 table fields, 26 classes, 152 methods and 53 test cases. We injected faults in the source code and then executed the test suite. A total of 15 fault injections were prepared each of which injected a single fault into a single production code query.

Tarantula provides a high level view of suspicious statements. Suspiciousness (Susp) is computed by the following equation:

\[
\text{Susp}(e) = \frac{\text{failed}(e)}{\text{totalfailed}} \times \frac{\text{passed}(e)}{\text{totalpassed}} + \frac{\text{failed}(e)}{\text{totalfailed}}
\]

But we need to find the suspicious queries (SuspQuery):

\[
\text{SuspQuery}(e) = \begin{cases} \text{Susp}(e), & \text{if query}(e) \\ \text{undefined}, & \text{otherwise} \end{cases}
\]

where \(\text{query}(e)\) is true when \(e\) is between 0 to 1 and false otherwise.

The application source code contains different types of statements, out of which we only need query statements to solve our problem. For that reason we explore Tarantula to see how much help this tool can provide. Our initial hypothesis was that Tarantula could accurately identify the faulty query statements by ranking them as highly suspicious. To explore this hypothesis, we executed our test suite on the fault injected source code to see if Tarantula returned enough information to retrieve faulty queries.

We need to find a set of candidate faulty queries to work with, which are embedded into the statements in a program \(P\). However, Tarantula can find suspicious statements but this is not the same as what we want. We have to separate query statements from the other non-query statements in program \(P\) to diagnose the faults in the queries. Tarantula alone might not be the ideal solution. Therefore, we need to collect and analyse information about the following questions to see how far Tarantula can be used in solving our problem:

**Q1:** Can we use the Tarantula suspiciousness metric to identify a set of candidate faulty queries? and

**Q1.1:** Does the Tarantula suspiciousness metric always rank faulty queries as highly suspicious? and

**Q1.2:** Does the Tarantula suspiciousness metric rank many non-faulty queries as highly suspicious? and

**Q1.3:** Does the Tarantula suspiciousness metric rank many (or any) non-query statements as suspicious when the only fault in a program exists in a query?

4.2 Experiment Results

This section presents main results of the experiment with Tarantula in Table 1, which shows the number of queries assigned to each colour (in decreasing order of suspiciousness i.e., Red, Orange, Yellow, Green and White) by Tarantula. The colour of the actual faulty query statement (FQSC) is also given.

<table>
<thead>
<tr>
<th>ExpID</th>
<th>R</th>
<th>Q</th>
<th>O</th>
<th>Y</th>
<th>G</th>
<th>W</th>
<th>FQSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td></td>
<td>Orange</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>Yellow</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Orange</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>14</td>
<td>0</td>
<td></td>
<td>Yellow</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td></td>
<td>Orange</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td></td>
<td>Orange</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>32</td>
<td></td>
<td>Orange</td>
</tr>
</tbody>
</table>

As our results show that Tarantula ranks faulty queries as highly suspicious in all the cases so Tarantula’s metric is helpful for our question Q1.1. For the next question, Q1.2, none of the experiments revealed any non-faulty query which Tarantula ranked as suspicious (these results are not shown in above table, that they have been omitted for lack of space). In case of the last question, Q1.3, our experiments found a few non-query statements ranked as highly suspicious whereas the faults were injected in query statements only. But these statements were generally related to the query statements, such as expressions with query strings, result-sets and decision statements or fields and their initialization.
However, for question Q1, we have found from our results that Tarantula is helpful as this technique assigns suspiciousness measures to each of the statement, including the query statements, in a program. In more than half of the experiments there was exactly one statement which was ranked as highly suspicious (i.e., marked red) and that statement was the actual faulty statement. In over 40% of the experiments, one statement was given the next highest suspiciousness ranking (i.e., orange) and that statement was actually the faulty statement. In the rest of the experiments, the faulty statements were ranked next higher level by marking them as yellow. So these results for finding the faulty statements including the query statements are encouraging.

Based on these results we can see that Tarantula can be used to identify suspicious query statements but we would need to develop a component which can filter out non-query statements to find sets of candidate queries from the Tarantula scores for our further analysis of the problem.

5. OUR APPROACH

In this section we describe our approach to addressing our research questions identified in section 2.1.

We plan to look initially at three specific kinds of fault (“fault hypotheses”) which are:

FH1: simple projection faults, where columns on a single table are projected incorrectly. That is, a query which projects out columns ProjCols from table T is incorrectly projecting some subset of these columns, FaultProjCols. Instead, the query should project (ProjCols ∖ FaultProjCols) ∪ CorrProjCols.

FH2: multiple table projection faults, where columns from several tables (e.g. from a join of two tables) are incorrectly projected, as described for FH1.

FH3: missing tuple selection fault, where a query over a single table has too strong a selection condition, resulting in one or more tuples being omitted from the query result. We will look specifically at off-by-one faults, caused by using < when ≤ is intending, and vice versa, and use of > when ≥ is intended and vice versa.

By considering these specific fault types, we hope to move towards a more generic method.

Our plan also contains tasks to evaluate the work and to assess impact. Evaluation will involve fault injection, and assessment of the ability of our system to correctly identify the injected faults, and to identify only a few non-existent faults. We will also need to measure the time required to produce the diagnosis, as this will be a factor in determining usefulness of our result. By “assess impact” we mean that we will need to look at how much our simplifying assumptions reduce the usefulness of our results in practice.

The plan below lists the tasks to be completed which are listed in roughly the order we intend to tackle them in.

Task 1. Find Candidate Responsible Queries (RQ 1 section 2.1)

We are developing a component that can, given the source code of a software system S and a test suite TS for that system, produce a report listing for each failed assertion the queries that could be a fault. This report is intended to be viewed by members of the development team (e.g. in an overnight e-mail message from the continuous testing system). Essentially, we will be listing all the queries in the trace for each test case produced by Cobertura3, ordered by descending suspiciousness as computed by Tarantula.

Task 2. Produce Fault Hypotheses (FH1) for Candidate Responsible Queries (RQ 2)

In this task, the component produced in task 1 will be extended to examine each failed assertion/candidate responsible query pair to see whether the failed assertion might be caused by a fault of type FH1 in the candidate query. This involves working out what the conditions that indicate the presence of a fault of type FH1 are, and writing a component to test them. E.g., given a failed assertion: assertEquals(50, hotelRoomType.getRoomRate()); which fails with actual value 45, and expected value 50. There show a query that is in the trace for this: “SELECT minRoomRate FROM hotelRoomType WHERE roomTypeID = ” + this.roomTypeID; The component for FH1 would check whether any other column of the tuple returned has the expected value 50.

The result of this version of the system is a report that gives additional information for those candidate queries that may contain FH1 faults, describing the columns that are wrongly projected, and stating what columns should be projected instead. The evidence for this proposal should also be reported.

Task 3. Apply Fault Hypotheses (FH1) and See Effect on Test Results (RQ 3 and RQ 3.1)

In this task, the component produced in task 2 is combined with a new component that applies the changes implied by the fault hypotheses (of type FH1) produced in task 2, and executes the test suite on the modified component to see whether the change implied by the fault hypothesis really does correct the failed assertion at which it is directed.

Task 4. Evaluate Component Handling FH1 (RQ 4)

We need to run two forms of evaluation: performance and correctness.

For both, we will need to inject faults (probably using a standard mutation tool, adapted so that it makes changes only to database queries[19]), and then assess how the system behaves with regard to each injected fault. We will count CPU time/elapsed time, plus true positives/false positives/false negatives.

Task 5. Interface for User to Select Delta to Apply Permanently. (RQ 5)

This task implements the code to finish the user process, by providing a facility for the user to request that a specific change be added to the code permanently, e.g. by clicking on a hyperlink in the text of the report that indicates which change is to be applied.

The same sequence of tasks (Task 6 to Task 11) will be performed for the remaining fault hypothesis types (FH2 and FH3) and the results illustrated.

Task 12. Assess Impact

In this task, we will look at some open source systems with tests of the data level and embedded queries to see how often our simplifying assumptions hold within real code, and therefore to get some idea of how useful our approach is in

3Cobertura is an open source coverage tool - http://cobertura.sourceforge.net/
practice.

We intend slotting in this task as soon as is convenient -though we may have to revisit the work in the light of lessons learnt after completion of tasks 2 (FH1), 6 (FH2) and 9 (FH3).

Task 12. Work with ConQueR

After writing components/designing techniques to handle the 3 specific fault types, we will look at creating a generic broker component that takes a variety of (simple or complex) fault hypothesis components, and applies them as seems appropriate. At the simplest end is our FH1, and at the other systems like ConQueR.

6. CONCLUSIONS

We have identified a problem for diagnosing faults in embedded queries that has yet not been tackled. We gave a description of this problem and surveyed the literature to find the current state of research in this field. We found that the existing techniques provide a partial solution to our problem. We performed an experiment with Tarantula which ranks suspiciousness of all the faulty statements in a program P but we need only faulty query statements to solve our problem. Therefore, Tarantula is partially helpful in identifying the faulty query statements and we still need to separate suspicious query statements from the other suspicious statements. We have started work to develop a component to find such candidate queries. This component will be used to see if the failures are caused by faults of specific types. It will be further extended to examine the effects of such faults on the test results which can be presented to user for making permanent change after satisfactory evaluation.

7. ACKNOWLEDGMENTS

The author gratefully acknowledges the support and encouragement of his supervisor, Dr Suzanne Embury, who also supplied the code for the experiments.

8. REFERENCES