BQL: Capturing and Reusing Debugging Knowledge [*]

Zhongxian Gu Earl T. Barr Zhendong Su
Department of Computer Science, University of California at Davis
{zgu,etbarr,su}@ucdavis.edu

ABSTRACT

When fixing a bug, a programmer tends to search for similar bugs that have been resolved in the past. A fix for a similar bug may help him fix his bug or at least understand his bug. We designed and implemented the Bug Query Language (BQL) and its accompanying tools to help users search for similar bugs to aid debugging. This paper demonstrates the main features of the BQL infrastructure. We populated BQL with bugs collected from open-source projects and show that BQL could have helped users to fix real-world bugs.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Reliability

Keywords

BQL, Reusing debugging knowledge

1. INTRODUCTION

Some bugs, among the millions that exist, are similar to each other. When a programmer encounters a bug, it is likely that a similar bug has been fixed in the past. A fix for a similar bug can help the programmer understand his bug, or even directly fix his bug. Studying bugs with similar symptoms, programmers may determine how to detect and resolve them. This is why programmers often search for similar, previously resolved, bugs.

Leveraging past solutions to current problems improves programmers' productivity. The solutions, in the form of a variety of formats, are stored on diverse systems, such as bug tracking systems, SCM commit messages, and mailing lists. The central challenge here is how to efficiently and accurately search and effectively reuse this large and unwieldy set of data sources.

Existing work has focused on analyzing bug reports and other unstructured bug information. DebugAdvisor combines both structured and unstructured data such as natural language description, debug output, and stack trace to construct a new sort of query, called fat query [1]. DebugAdvisor returns large result sets that require human processing. Jula *et al.* proposed Dimmunix, a system that prevents previously encountered deadlocks from happening again in a program [3]. Dimmunix stores previous deadlock patterns, a method invocation sequence along with thread scheduling decision, and monitors the current system to prevent it from being trapped into those deadlocks again. Dimmunix targets only deadlock bugs.

A common practice when bug-fixing is to paste keywords into a search engine, like Google. A search engine searches the large data sources with keywords, resulting in many false positives. Searches predicated on incorrect keywords might further pollute the results. This paper describes the bug query language (BQL) infrastructure, designed from the ground up to facilitate the search for similar bugs, and demonstrates how to use it to aid debugging. At the heart of this infrastructure is BQL, a flexible query language that allows users to express a wide variety of queries over trace comparison. Compared with a search engine, BQL focuses on the systematic use of bug semantics — BQL searches for bugs similar to a queried bug by comparing execution traces of the queried bug with execution traces of other bugs.

To use BQL, users must populate its database with their bugs. The BQL infrastructure provides tools for collecting execution traces from bugs and uploading them into its database. Users then issue queries to search for similar bugs. Users can explore their queries' result sets to determine whether their behavior and fixes help fix an open bug. We have a technical report [2] that focuses on validating our hypothesis that unique bugs are rare, and the detailed design and implementation of BQL and its supporting infrastructure.

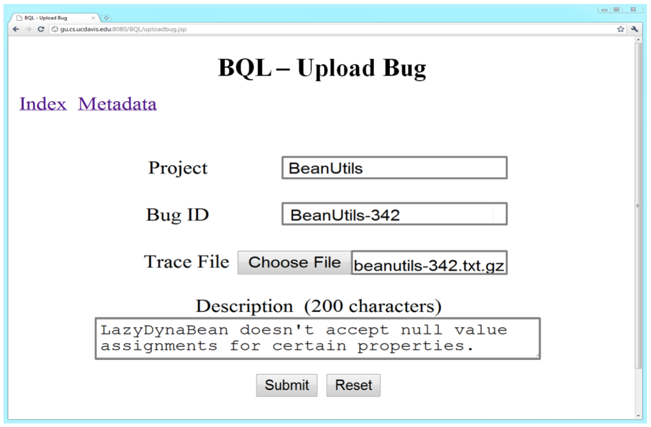
To show how to use the BQL infrastructure, we have populated its database with bugs from `Rhino` and the Apache `Commons` projects. Section 2 catalogs the features of BQL. Section 3 presents two detailed scenarios, drawn from real bugs in the Apache `Commons` projects, that demonstrate how BQL can speed debugging. BQL is open to the public at `http://gu.cs.ucdavis.edu:8080/BQL`.

2. MAIN FEATURES OF BQL

The BQL infrastructure consists of an offline trace collection framework and a web-based user interface that allows user to upload their execution traces, search for and explore similar bugs.

Collecting and Uploading Traces. To use BQL, a user needs to populate BQL's database with his bugs. BQL uses execution traces to index bugs, so a user must collect and upload them. BQL's trace collection framework provides a tool that instruments class and jar files in a directory. After instrumenting a program, the user runs it against a bug-triggering input to generate a trace file, which he then uploads into BQL, as shown in Figure 1. In addition to uploading

[*]This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the U.S. Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.



BQL – Upload Bug

Index Metadata

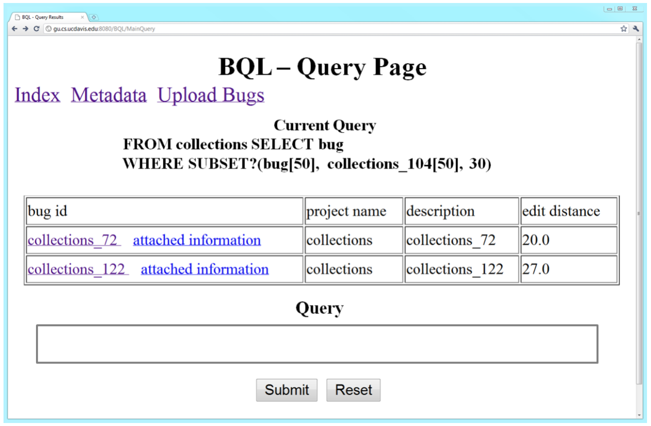
Project	BeanUtils
Bug ID	BeanUtils-342
Trace File	Choose File beanutils-342.txt.gz

Description (200 characters)

```
LazyDynaBean doesn't accept null value
assignments for certain properties.
```

Submit Reset

Figure 1: To upload a bug, the user selects (or adds) a project, accepts (or overrides) the default id, then chooses the trace file.



BQL – Query Page

Index Metadata Upload Bugs

Current Query
FROM collections SELECT bug
WHERE SUBSET?(bug[50], collections_104[50], 30)

bug id	project name	description	edit distance
collections_72 attached information	collections	collections_72	20.0
collections_122 attached information	collections	collections_122	27.0

Query

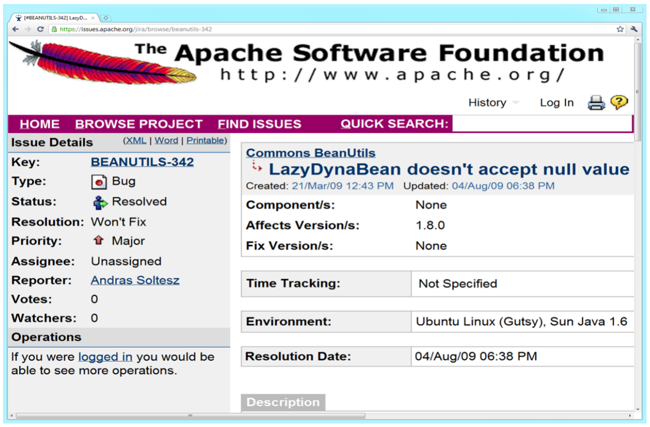
Submit Reset

Figure 2: To issue a query, the user types the query in the text box and clicks submit.

the trace file, this page asks the user to select or add a project name and allows the user to add a brief description. Using the project name, BQL suggests an id, as shown. After a successful upload, BQL opens a page that displays the uploaded trace. BQL currently contains 546 bugs collected from Rhino and the Apache Commons projects. We invite users to upload their bug traces, use BQL, and contribute to it.

Issuing Queries. The syntax of BQL is based on the standard query language SQL. The details of its syntax and semantics are described in the technical report [2]. Here, we illustrate BQL with simple examples. Please refer to our online documentation for more extensive tutorials. We begin with the simplest database query: returning all rows in the database. In BQL, this is "`SELECT` *bug* `FROM ALL WHERE TRUE`". To return all the bugs in one project, say `Rhino`, we replace `ALL` with `rhino` to return all `rhino` bugs in the database and issue "`SELECT` *bug* `FROM rhino WHERE TRUE`".

Traces may differ in numerous ways irrelevant to a bug's semantics. For example, two traces may have taken different branches of a conditional when neither branch has any relation to the bug. As another example, trace events with the same semantics in a particular context may have different names. Concrete execution traces can therefore obscure semantic similarity, both cross-project and even within project. To combat the false negatives this can cause, BQL allows two traces to differ in either Hamming or Levenshtein edit distance, with Levenshtein the default. BQL also supports user-defined distance functions, adapted to a particular problem domain.



The Apache Software Foundation
http://www.apache.org/

History Log In

HOME BROWSE PROJECT FIND ISSUES QUICK SEARCH:

Issue Details (XML | Word | Printable)

Key:	BEANUTILS-342
Type:	Bug
Status:	Resolved
Resolution:	Won't Fix
Priority:	Major
Assignee:	Unassigned
Reporter:	Andras Soltesz
Votes:	0
Watchers:	0

Operations

If you were logged in you would be able to see more operations.

Commons BeanUtils
LazyDynaBean doesn't accept null value
Created: 21/Mar/09 12:43 PM Updated: 04/Aug/09 06:38 PM

Component/s:	None
Affects Version/s:	1.8.0
Fix Version/s:	None
Time Tracking:	Not Specified
Environment:	Ubuntu Linux (Gutsy), Sun Java 1.6
Resolution Date:	04/Aug/09 06:38 PM

Description

Figure 3: BQL links this page in the Jira bug tracking system.

Appending a DISTANCE clause to a query overrides the default, *e.g.* "`SELECT` *bug* `FROM ALL WHERE TRUE DISTANCE HAMMING`".

Bugs may be similar in different ways. Thus, the query language allows users to define their own notions of trace similarity. BQL provides powerful operators that give users flexible control over trace comparison. When we are debugging, we are often interested in the behavior of a program shortly before an error manifested itself. BQL allows searching traces for bugs with similar suffix traces

$$\texttt{SELECT } \mathit{bug} \texttt{ FROM Collections}$$
$$\texttt{WHERE SUBSET?}(\mathit{bug}[50], \texttt{Collections-104}[50], 30).$$

Here, `Collections-104` is the queried bug against which BQL searches for similar bugs. [50] is the suffix operator that prunes traces to suffixes of length 50. `Collections-104`[50] is the set of length 50 suffixes of the traces in `Collections-104` and *bug*[50] is the set of length 50 suffixes for an arbitrary bug. The `FROM` clause restricts the search to bugs in the `Collections` project.

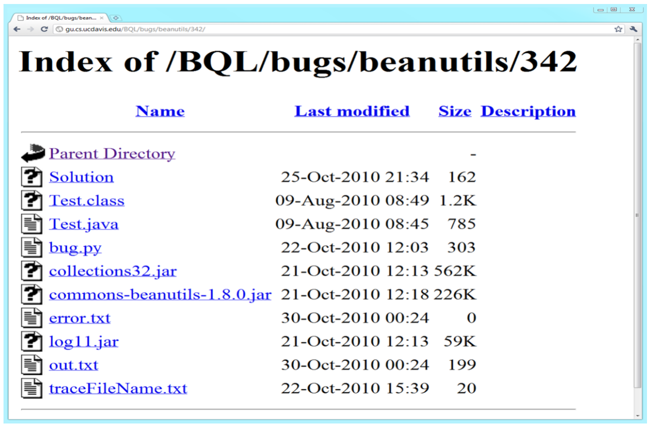
This query returns those bugs in the `Collections` project whose length 50 suffixes are a subset of those of `Collections-104` when up to 30 Levenshtein edits are applied. Figure 2 shows the results of the query. For usability, BQL returns similar bugs in order of increasing edit distance so that users can put more effort into analyzing bugs whose traces are closer, in edit distance, to the queried. BQL allows issuing a query that matches against the previous result set. This allows users to write a sequence of queries that gradually refine their results. To do this, users issue a new query on the query page without resetting. To begin a fresh query, they simply reset before issuing the query.

When asking whether two bugs are similar, a user may wish to project irrelevant events out of the trace before the comparison. For example, when searching for bugs similar to the bug `527`, a user may want to drop methods in the `log` package to reduce noise. The query

$$\texttt{SELECT } \mathit{bug} \texttt{ FROM ALL}$$
$$\texttt{WHERE SUBSET?}(\texttt{PROJ}(\mathit{bug}, \texttt{\^{}log}), \texttt{527}, 10)$$

accomplishes the task. In this query, `PROJ`(*bug*, `^log`) removes all the methods in the `log` package from traces *bug*, then asks whether a candidate bug's set of traces is a subset of those of `527` with up to 10 edits allowed. In addition to its `SUBSET?` predicate, BQL defines an `INTERSECT?` predicate that returns true when a candidate bug's set of traces has a nonempty intersection with the queried bug's traces.

Exploration. To be useful, the BQL system must make it convenient to explore each bug in a result set. BQL provides two ways



Index of /BQL/bugs/beanutils/342

Name	Last modified	Size	Description
Parent Directory		-	
Solution	25-Oct-2010 21:34	162	
Test.class	09-Aug-2010 08:49	1.2K	
Test.java	09-Aug-2010 08:45	785	
bug.py	22-Oct-2010 12:03	303	
collections32.jar	21-Oct-2010 12:13	562K	
commons-beanutils-1.8.0.jar	21-Oct-2010 12:18	226K	
error.txt	30-Oct-2010 00:24	0	
log11.jar	21-Oct-2010 12:13	59K	
out.txt	30-Oct-2010 00:24	199	
traceFileName.txt	22-Oct-2010 15:39	20	

Figure 4: Bug data cached by BQL.

```
1  DynaBean myBean = new LazyDynaBean();
2  myBean.set("myDynamicKey", null);
3  Object o = myBean.get("myDynamicKey");
4  if ( o == null )
5      System.out.println("o is null.");
```

Figure 5: `LazyDynaBean` does not return null.

to explore a bug — linking the original bug data in a bug tracking system, such as Bugzilla or Jira, and caching some of that data locally. In Figure 3, BQL links the bug page in the original bug tracking system. This page contains discussions among developers and reporters and ancillary material. These pages can be quite large; it might take some time for a user to scan them in their entirety to discover a solution. To save time, the BQL infrastructure itself stores the buggy source code, the fix and failing test case (if available), and useful comments gathered during bug collection. Users can explore a bug's directory, shown in Figure 4, to find a solution more quickly. In the future, we will extend BQL to allow users to update this data.

3. USAGE SCENARIO

In this section, we use two real bugs from the Apache Commons projects to construct two scenarios that demonstrate how BQL can help a programmer to fix his bug by identifying similar bugs.

The Apache Commons `BeanUtils` project provides easy-to-use wrappers for dynamic Java beans. In particular, the `BeanUtils` project allows a programmer to instantiate a `LazyDynaBean` to lazily set and get values without statically knowing the property name of a Java bean, as on line 2 of Figure 5. The bug `BeanUtils-342` was filed concerning the behavior that `LazyDynaBean` does not return null when a property is explicitly set to null.

How could BQL have helped the reporter of `BeanUtils-342` solve this problem? Assuming BQL's database had been populated with traces from `BeanUtils`, the bug reporter could issue queries to search for similar bugs to `BeanUtils-342`. When the reporter does not deeply understand the problem, matching trace suffixes is a natural way to search for bugs, since many bugs cause termination. The reporter tries the following query:

`SELECT` *bug* `FROM BeanUtils`
`WHERE SUBSET?`(*bug*[50], `BeanUtils-342`[50], 50).

This query returns bugs in `BeanUtils` whose length 50 suffixes are a subset of those of `BeanUtils-342` when up to 50 Levenshtein edits are applied.

The query returns `BeanUtils-24` first with distance 34, whose problem appears identical to that of `BeanUtils-342`: the `get` method of `LazyDynaBean` did not return null when the property was explicitly set to null. Studying the solution to `BeanUtils-24` reveals that the observed behavior is the intended behavior and provides a workaround: subclass `LazyDynaBean` class and override the `CreateProperty` method to return `null`. The reporter realizes that he can also apply the similar workaround to fix his bug: subclass `LazyDynaBean` class and override `CreateOtherProperty` method to return null. In fact, a project developer replied with the same fix five months later after the reporter filed `BeanUtils-342`. With the help of BQL, the reporter could have resolved the bug in minutes, instead of possibly waiting five months for the answer.

Our second example involves the Apache Commons `Configuration` project, which defines an interface for reading configuration data from a variety of sources such as XML or property files. Bug `Configuration-323` occurred when the `DefaultConfigurationBuilder` class misinterpreted property values as lists when parsing configuration files. To gain insight into this bug, the bug reporter could have tried to search for similar bugs using suffix comparison again. The query

`SELECT` *bug* `FROM Configuration`
`WHERE SUBSET?`(*bug*[50], `Configuration-323`[50], 50)

returns `Configuration-354` first, at distance 26 from `323`. In this bug, `XMLConfiguration` mishandled a non-default list delimiter. The fix of `354` is not helpful; the suffixes of `323` and `354` are similar because their test cases both process `XML` configuration files.

In the bug report of `Configuration-323`, the reporter speculated that the invocation of `ConfigurationUtils.copy` during internal processing might be the cause. To confirm this hunch, the reporter might use BQL's regular expression matching to investigate all bugs that call `ConfigurationUtils.copy`. The query

`SELECT` *bug* `FROM Configuration`
`WHERE SUBSET?`(*bug*, `ConfigurationUtils.copy`)

searches for bugs in the `Configuration` project that invoke `ConfigurationUtils.copy`. The result set contains `Configuration-272` and `283`. Studying these bugs reveals that calling `ConfigurationUtils.copy` caused both of them and that project developer provided a workaround. These bugs predate `Configuration-323`. Thus, the reporter could have used BQL's pattern matching to resolve `Configuration-323` by identifying and studying `272` and `283`, then applying their workaround.

4. CONCLUSION

We have described the BQL infrastructure that compares execution traces to find similar bugs to aid debugging. We have demonstrated how to collect and upload traces, issue queries, and explore result sets in BQL. We have also shown how BQL can be used to aid debugging. BQL is open to the public at `http://gu.cs.ucdavis.edu:8080/BQL`. Our demonstration video is available at `http://www.youtube.com/watch?v=yzm9iD5Ow9w`. We invite readers to use BQL and help us improve it.

5. REFERENCES

[1] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A recommender system for debugging. In *ESEC/FSE*, 2009.

[2] Z. Gu, E. T. Barr, and Z. Su. Language and tool support for semantic bug analysis. Technical Report CSE-2011-3, Department of Computer Science, University of California, Davis, 2011.

[3] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.