Statistically optimize mysql-test runs by running less tests

Student: Pablo Estrada
Project: MariaDB
Email: polecito.em@gmail.com
Residence: South Korea
Phone number: +82 010 9746 9747

Abstract

Software engineering relies heavily on thorough testing. This is especially true for systems programming. The goal of this project is to produce a small piece of software that will leverage historical testing data from mysql-test runs on MariaDB, so that instead of using a 'brute force' approach and running as many test suites as possible, we can intelligently select a smaller set of test suites that are more likely to find defects.

Self-introduction

My name is Pablo Jose Estrada Murguia. I am from **Mexico**, and currently pursuing a **master's degree** in **South Korea**.

I did my bachelor's degree in **Computer Engineering**, in the National University of Mexico. During my degree, I also attended the University of California in Los Angeles (UCLA), and Stanford University as an exchange student.

Upon graduation from university, I started working for Oracle Corporation. I worked there for 2 and a half years, coding **mostly in C and PL/SQL**, as well as some **Java**; committing code to the Oracle executable. Particularly, for the DBMS_SCHEDULER functionality of Oracle. I worked at Oracle from January 2011 to June 2013.

My **experience contributing to open source software** is very recent. To learn Korean I use an open-source flashcard application called **AnkiDroid**. I decided to contribute, and started a few months ago. I have **committed a few transactions** to the code, running in the new release. I should say that the feeling is very satisfying. I'm still new to open-source, but I'm quite excited about it. My contributions to AnkiDroid are **available on my github**, referenced at the end of the proposal.

Why am I a good match for the project

Although I have long been interested in computer science, and software development; while I was working at Oracle I found that research using **data and code together** was much more satisfying than pure software-making.

That's why recently I try to reduce my involvement in pure software engineering, and move closer to the data-side of it.

I also believe that my experience at Oracle working directly **inside a database** and using **automatic testing**, prepared me to understand **the context** in which the project will be useful.

Oracle is an excellent example of a place where smarter testing can reduce costs and turnout time on the humongous server farms that spend 24 hours-a-day crunching database queries and test suites.

Background

Testing is one of the most important and time-consuming features of software development. Particularly in systems programming there is a constant struggle between keeping the stability of a codebase and making progress in new features and bug fixing.

Unfortunately, with more code and more complexity in a software system, it becomes more difficult to foresee and retrieve all the unintended consequences of every code change; and we end up needing to rely on thorough and extensive testing.

However, testing requires time and resources, both of which are limited. That's why it is important to try to test optimally and smartly. Here is where data can play an important role.

Throughout the software development process, extensive data is generated and stored. Data regarding the code base, the test suites, the behavior of the codebase in tests, etc. All these data had only been gathered for the sake of archiving and management; but up until now, this kind of data had not been used to extract new and deeper knowledge about our codebase.

The goal of this project is to utilize this data to optimize the resources that we use in testing. Looking at correlations, and patterns inside these these data we expect to be able to extract the knowledge to allow us to test more smartly and predict the best set of tests to maximize likelihood of thorough testing of a change and its effects.

Proposal

Goals

- Design a system that automatically aggregates and **analyzes** data from the **MariaDB codebase** and **mysql-test run history**.
- Design a system that using this aggregate information, can **analyze a code transaction**, and **output** a **set of recommended tests**.

Inputs

- Information of a code transaction (files changed, diffs, functions changed, etc)

Outputs

- Recommended set of tests to run

The expected result of this project is a small system in the form of a **script** that can be run on the **buildbot system**; and the script should return a recommended set of tests to run for the build.

Steps

1. Review and organize all the data available (test results history, build history, change

history on code files, latest transaction…).
2. Depending on the data available, decide on an algorithm to aggregate and organize data.
3. Decide on the supervised learning algorithm, also considering the data available.
4. Use first ~80% of data points as the training dataset, and test the algorithm on the latter 20% of data points. Tune algorithm if necessary.
5. If algorithm performance is satisfactory, implement on Buildbot; if not, revise the supervised learning algorithm and go back to step 3.
6. Test prediction algorithm on live builds. If results are satisfactory, finish project. If not, revise.

Relevant data

- Inter-test diff/failure/no failure correlations
- Individual test failure rate
- Correlation between file-changes and test failures (Is this data available or easy to infer?)
- Number of bug-fixing transactions applied to a file
    - If a file has a lot of bug-fixing transactions, testing should be more thorough upon modification.
- Dependencies inside the code base
    - Measured in the codebase through file inclusion graphs, function call graphs…
    - Measured in the code history through correlations of occurrence in transactions
    - Measured in test history
- The data should also be **weighted** inversely on time (newer data should be weighted more heavily than older data)

Technical introduction

The data mining algorithm should look into the version tree provided by the buildbot and extract the information only from branches that are relevant to the current revisions. It is important to avoid using data of different or outdated forks to try to forecast failures in the updated codebase.

The buildbot database contains the following tables and data structures:

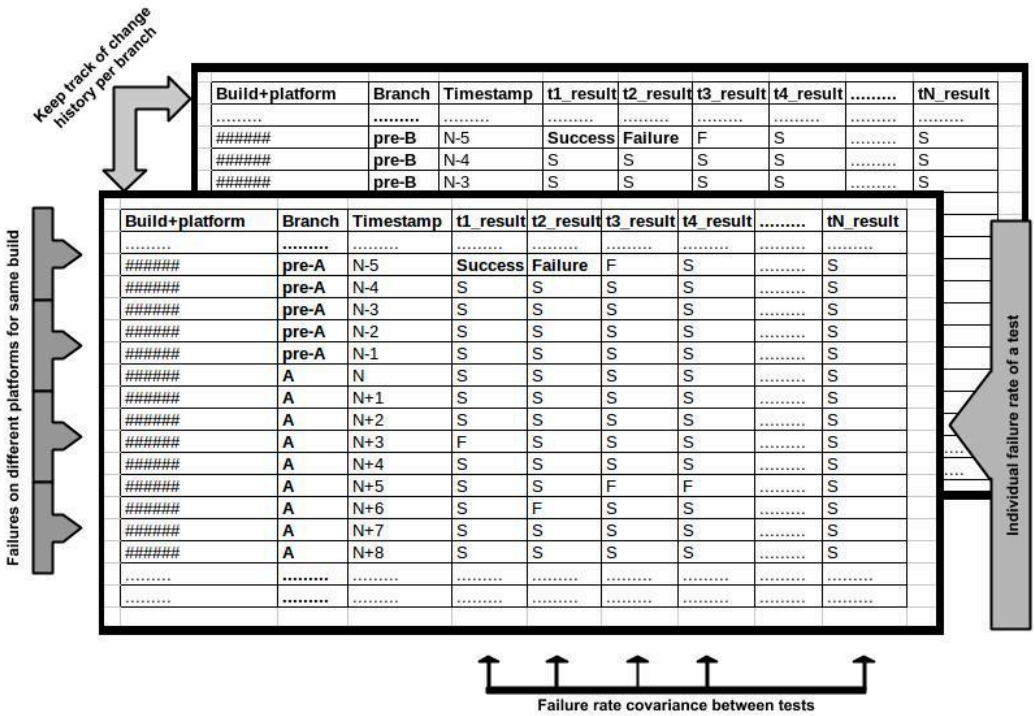
The ***test_run*** table, which contains information regarding the run of each test. In this table the most relevant fields for the project are the following:

- ***platform*** - This is the platform on which the build and test were run.
    - The *platform* variable is an important confounding variable. There are platforms that are specially prone to display one issue or another. This should be accounted for.
- ***branch*** - This is the unique identifier of each branch
    - The *branch* variable is important to keep track
- ***bt*** - Date when the buildbot run was started. This is important because we want to give

relevance to the most recent data.
    - The timestamp is a linear variable -UNIX epoch time. This is perfect for the exponential decay utilized in [2].
- ***bbnum*** - This is a build identifier. Along with *platform*, it identifies a build in buildbot uniquely.
- ***test_name*** - This is the unique identifier of each test.
    - Also, the *typ* field specifies whether a test has particular conditions that should be noted

The ***test_failure*** and ***test_warning*** tables contain information of every test run that showed any kind of anomaly. In an initial analysis, it seems that **whether or not a test failed** should be the only factor to consider.

The following diagram shows what the test history dataset looks like, and points out some of the factors that should be considered when analyzing the dataset.



Build+platform	Branch	Timestamp	t1_result	t2_result	t3_result	t4_result	.....	tN_result
.....	.....	.....	.....	.....	.....	.....	.....	.....
######	pre-B	N-5	Success	Failure	F	S	.....	S
######	pre-B	N-4	S	S	S	S	.....	S
######	pre-B	N-3	S	S	S	S	.....	S

Build+platform	Branch	Timestamp	t1_result	t2_result	t3_result	t4_result	.....	tN_result
.....	.....	.....	.....	.....	.....	.....	.....	.....
######	pre-A	N-5	Success	Failure	F	S	.....	S
######	pre-A	N-4	S	S	S	S	.....	S
######	pre-A	N-3	S	S	S	S	.....	S
######	pre-A	N-2	S	S	S	S	.....	S
######	pre-A	N-1	S	S	S	S	.....	S
######	A	N	S	S	S	S	.....	S
######	A	N+1	S	S	S	S	.....	S
######	A	N+2	S	S	S	S	.....	S
######	A	N+3	F	S	S	S	.....	S
######	A	N+4	S	S	S	S	.....	S
######	A	N+5	S	S	F	F	.....	S
######	A	N+6	S	F	S	S	.....	S
######	A	N+7	S	S	S	S	.....	S
######	A	N+8	S	S	S	S	.....	S
.....	.....	.....	.....	.....	.....	.....	.....	.....
.....	.....	.....	.....	.....	.....	.....	.....	.....

The information regarding testing of previous builds can provide a lot of insights, but I am particularly interested on **leveraging another important set of data**: The data provided by **Bazaar** whenever a new transaction is to be tested. This information is available to **Buildbot** as

part of its **change source**, and should help us run a set of tests specially selected for the new set of changes in the codebase. This is the first area that I will look at once I get access to the Buildbot master databases and repository database.

Algorithms to consider

The following are **supervised learning** algorithms that are good candidates to be used in the fault prediction system. They are presented in order of complexity. Depending on the set of available data

- Logistic regression - Logistic regression is a good, simple candidate. If we are only considering test history -and maybe test results correlation-, logistic regression would seem like a good enough algorithm to consider.
- Bayes classifier - A Bayes Classifier is a simple enough model that is able to incorporate several data dimensions into analysis.
- Neural networks/Decision trees/Random forests - These three methods are good candidates to leverage the large volume of data available: They all can take into consideration an individual test's history, along with its correlation with other tests and it is also possible to include transaction information such as files modified since last run, and file change history.

Idea

Using all the available data, the script should be able to analyze a transaction and condense in a single number **an estimate of the relevance of each test to the transaction**. If the relevance of a test is **higher than a threshold**, then this test is recommended for execution.

The **threshold** should be **tuned** according to several factors: How **sensitive are the files** in the transaction (see [2]), average volume of testing that should be done per transaction, closeness to a release, size of a transaction, etc.

Useful literature

[1] Testing at the speed and scale of Google - Using code dependency graphs to select a set of tests to run - http://google-engtools.blogspot.kr/2011/06/testing-at-speed-and-scale-of-google.html

[2] Bug prediction at Google - Using bug history to recognize files that are more sensitive to changes - http://google-engtools.blogspot.kr/2011/12/bug-prediction-at-google.html

[3] GitHub - pabloem - https://github.com/pabloem/