Unit Tests for SQL

Spinellis, Diomidis

DOI
10.1109/MS.2023.3328788

Publication date
2024

Document Version
Final published version

Published in
IEEE Software

Citation (APA)
Spinellis, D. (2024). Unit Tests for SQL. *IEEE Software*, *41*(1), 31-34.
https://doi.org/10.1109/MS.2023.3328788

Editor: Diomidis Spinellis
dds@aueb.gr

Unit Tests for SQL

Diomidis Spinellis

I LEARNED TO program in an age where testing meant prodding a program (larger software systems were rare at the time) with various inputs to see whether it failed. We have come a long way since that time.

The first testing code I saw was in the BIOS (basic input/output system) of the original IBM personal computer in the early 1980s. More than 1,000 lines of assembly language instructions tested all the computer's hardware, starting with the Intel 8088 processor (flags, registers, and conditional jumps) and continuing with memory and peripherals.[1] Undoubtedly, decades of hard-earned experience had instilled into IBM's DNA the understanding that anything can fail and nothing should be assumed.

A few years later, when compiling the first release of the Perl scripting language[2] (a precursor of Python), I encountered another concept that amazed me: a program that contained code to test itself. A series of 51 tests verified many aspects of the language and its built-in functions. I realized their value in delivering a reliable product but didn't see their direct relevance in everyday software development.

Digital Object Identifier 10.1109/MS.2023.3328788
Date of current version: 20 December 2023

For that I'd have to wait some more years for JUnit to popularize unit testing,[3] and for opportunities for me to work with teams that followed and valued the practice. Getting myself to write unit tests, especially outside established projects that used them, took effort. I had to learn to write testable code, find how to install and run a testing framework in each language I used, and establish the discipline of unit testing all the tricky code. Once these things were in place, I got hooked, wondering how I could ever work without unit tests in place.

Consequently, when I found myself writing dozens of database queries for software analytics tasks, I looked for a way to unit test them. I could not find a corresponding solution, so I developed RDBUnit (relational database unit testing), a unit testing framework for relational database queries. It is available as an installable Python package on PyPI (Python package index) (https://pypi.org/project/rdbunit/) and as open source software on GitHub (https://github.com/dspinellis/rdbunit/).

Unit tests for RDBUnit consist of three parts, following the common Arrange-Act-Assert pattern.

Writing SQL Unit Tests

The unit tests for RDBUnit consist of three parts, following the common Arrange-Act-Assert pattern: a setup block that defines the names and contents of some database tables, the query to be tested, and a block providing the expected result. The input and output are specified as table contents. The input starts with a line containing the words `BEGIN SETUP`, while the results start with a line containing the words `BEGIN RESULT`. The input and output are specified by first giving a table's name, followed by a colon. The name may be prefixed by the name of a database where the table is to reside, followed by a dot. The next row contains the table's field names, separated by one or more spaces. Then come the similarly formatted table's data, which are terminated by a blank line if another table's definition will follow, or by the word **END** at the end of the corresponding section. RDBUnit automatically determines the type of each table field (integer, real number, date, time, time stamp, or Boolean value) based on its contents.

In the "Results" section, the table name is not specified if the tested query is a selection statement rather than a table or view creation. Note that both the input tables and the results need not contain all fields of the production database tables; all that is needed are the fields that appear in the query. The setup-query-results sequence can be repeated multiple times within a test file (each starting with a new **BEGIN SETUP** line) to test several aspects of the query, such as duplicate input or output records, an empty result set, or diverse join scenarios.

The following is a complete example of an SQL unit test:

```
BEGIN SETUP
sales:
month    revenue
March    130
April    50
END

BEGIN SELECT
SELECT MAX(revenue) AS max_revenue
   FROM sales;
END

BEGIN RESULT
max_revenue
130
END
```

This sequence defines the contents of the sales table, provides a query to obtain the largest monthly volume of sales from it, and finishes with the result's queried name and expected value.

The query to test is either specified inline, as in the preceding example, with a **BEGIN SELECT** (for selection queries) or a **BEGIN CREATE** (for view and table creation queries) statement, or, more commonly, by including an SQL source code file through a corresponding **INCLUDE SELECT** or **INCLUDE CREATE** statement. For instance, the following statement will include the table creation query residing in the file named *leader_comments.sql*:

```
INCLUDE CREATE leader_comments.sql
```

The included file's name also forms the test's name. The queries stored in files can be easily loaded and run by the production system (often also as prepared SQL statements for achieving performance gains), thus ensuring that the tested code exactly matches the one used in production. This approach satisfies NASA's valuable lesson: "Test as You Fly, Fly as You Test."[4]

Test Execution

RDBUnit is a command-line tool that allows its use in the widest possible set of cases, such as diverse operating systems and headless continuous-integration environments. Its typical use involves specifying on the command line the database engine to use (SQLite, mySQL, and PostgreSQL are currently supported) and one or more unit test files. When invoked in this way, RDBUnit will produce on its standard output SQL statements that, when fed into the corresponding database engine, will produce the test results formatted according to the Test Anything Protocol (TAP).[5] TAP is a tool and language-agnostic format that allows any test output producer, such as RDBUnit, to produce output that many existing consumers, such as test report generators and test result browsers, can use. The following is what a typical execution might look like– for successfully executing the three unit tests in the file **cc.rdbu** through the PostgreSQL database. (The **–t –q** *psql* command arguments configure it to output tuples without any adornments.)

```
$ rdbunit --database=postgresql cc.rdbu |
> psql –U db_user –t –q testdb
 ok 1 – cc.rdbu: tl.nl_commits_l_comments
 ok 2 – cc.rdbu: tl.l_commits_nl_comments
 ok 3 – cc.rdbu: tl.commits_with_comments
 1..3
```

RDBUnit also offers the **-- results** option to list the results of a unit test query on its standard output. These can then be manually verified and integrated into the unit test as the expected results.

By default, RDBUnit creates a test database to run the tests in isolation. The **--existing-database** option can be used to run the tests in the context of the database engine's current database, thus allowing the fetching of data from existing populated reference tables.

Under the Hood

RDBUnit is a Python script of roughly 500 lines of code. Its implementation's key ideas are a simple parsing mechanism, a class hierarchy for managing database engine differences, and the evaluation of test results through SQL code.

RDBUnit's input is a domain-specific language (DSL): a language geared toward a particular purpose (unit tests in this case), rather than general purpose computing. When dealing with expert users, DSLs can be a powerful software architecture element because they enable the flexible and extensible expression of sophisticated structures. However, implementing a DSL can be a tricky balancing act.[6]

At one extreme of implementation choices lies the adoption of an existing flexible data format, such as XML, YAML, JSON, or INI (initialization file). Following this data-centric approach minimizes the implementation effort through the use of existing parsing libraries at the expense of usability, which is hurt by the format's lack of flexibility. For example, in the case of RDBUnit, none of the corresponding choices would allow the readable expression of tabular data.

At the other extreme lies the implementation of a full-fledged language front end through a lexical analyzer and a parser based on a specified formal grammar. This approach demands a substantial implementation effort but rewards with a robust front-end furnished with solid error handling.

Given that a working good-enough tool is, in most cases, superior to a perfect one that is too hard to build, RDBUnit takes a middle ground between the two extremes employing three tricks to parse the unit test input specifications.

First, it uses regular expressions to recognize the input elements it reads. For example, RDBUnit identifies dates expressed in the yyyy-mm-dd format through the regular expression "\d{4}-\d\d-\d\d" (four digits, followed by a dash followed by two digits, followed by another dash, followed by another two digits). The approach isn't perfect (for example, it allows the invalid date 2023-13-32), but it's good enough for writing unit tests.

Second, RDBUnit employs a line-based input format, a simple state machine, and a single-element stack to keep track of what input it is processing and what should come next. The initial state, changes into *setup*, *sql*, or *result* when it encounters the corresponding BEGIN lines. A separate *table_columns* state handles the parsing of each input or result table's columns.

RDBUnit handles the three supported database engines through a simple two-level class hierarchy. A class for each engine inherits a generic *Database* class that provides default implementations shared by multiple subclasses. Each class contains methods for tasks such as creating and dropping the temporary test database or schema where tests will run, using an existing database, and representing some data types.

A working good-enough tool is, in most cases, superior to a perfect one that is too hard to build.

Although many (mainly object relational mapping) packages exist with the aim of abstracting database engine differences, none of them could handle all the required tasks. It was thus easier to simply express the tasks as tiny sub-class methods.

Third, RDBUnit outsources the handling of SQL code to the database engine. Specifically, after issuing SQL statements for creating and populating the required tables, it creates an SQL view whose contents are defined to be the result of the tested query statement. Thus, the following SQL code is generated for the provided unit test example:

```
-- BEGIN SETUP
DROP TABLE IF EXISTS sales;
CREATE TABLE sales(month VARCHAR(255),
  revenue INTEGER);
INSERT INTO sales VALUES ('March', 130);
INSERT INTO sales VALUES ('April', 50);
-- BEGIN SELECT
CREATE VIEW test_select_result AS
  SELECT MAX(revenue) as max_revenue
  FROM sales;
-- BEGIN RESULT
DROP TABLE IF EXISTS test_expected;
CREATE TABLE test_expected(max_revenue
INTEGER);
INSERT INTO test_expected VALUES (130);
```

Finally, RDBUnit, rather than reading the query's result and comparing it with the expected one, generates standard SQL code that performs the comparison, yielding directly the TAP-conforming result. The following query compares the number of records in the union of the computed results and the expected results against the actual number of records in the computed results and the expected results. If these numbers differ, then one of the two tables will have fewer or more records than the other, and therefore, the test should fail. Communicating the test's failure is accomplished by obtaining the appropriate TAP result: "ok 1" or "not ok 1."

```
SELECT CASE WHEN
 (SELECT COUNT(*) FROM (
  SELECT * FROM test_expected
  UNION
   SELECT * FROM test_select_result
 ) AS u1) = (SELECT COUNT(*) FROM
test_expected)
AND
 (SELECT COUNT(*) FROM (
   SELECT * FROM test_expected
   UNION
   SELECT * FROM test_select_result
 ) AS u2) = (SELECT COUNT(*) FROM
test_select_result)
THEN 'ok 1 - <stdin>'
ELSE 'not ok 1 - <stdin>' END;
```

This approach ensures that testing is performed using purely mechanisms natively provided by the specified database engine, without having Python's interpretation of the results influencing the unit test's outcome.

RDBUnit helped me uncover subtle bugs, which might have otherwise resulted in erroneous findings.

Lessons Learned

I've used RDBUnit to write tens of SQL unit tests for tasks involving the identification of enterprise open source software contributions,[7] repository deduplication,[8] document processing, and citation analysis. In some cases, RDBUnit helped me uncover subtle bugs, which might have otherwise resulted in erroneous findings. In several other cases, expressing the unit test in terms of very simple input and expected results allowed me to concentrate on the essence of the query, quickly experimenting with several approaches to find the one that was correct, rather than wasting time waiting for the query to run on the full dataset and then painstakingly combing the obtained results for errors. If I had to write a complex SQL query on big data without RDBUnit at hand, I would quickly whip up a similar tool to work with the peace of mind offered by unit testing's guardrails.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business, Greece, and a professor of software analytics in the Department of Software Technology in the Department of Software Technology at the Delft University of Technology, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.

References

1. *IBM Personal Computer Hardware Reference Library: Technical Reference*, Revised ed. IBM Corporation, Boca Raton, FL, USA, Apr. 1984, pp. 5-33–5-49. [Online]. Available: https://archive.org/details/IBMPCIBM5150-TechnicalReference6322507APR84/page/n125
2. L. Wall, "Perl, a 'replacement' for awk and sed," *USENET Newsgroup Comp.sources.unix*, vol. 13, no. 1, Feb. 1, 1988. [Online]. Available: https://groups.google.com/g/comp.sources.unix/c/Njx6b6TiZos/m/X-JaOCXhPrsJ
3. K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Rep.*, vol. 3, no. 7, pp. 37–50, 1998.
4. "Test as you fly, fly as you test, and demonstrate margin (1998)," NASA Public Lessons Learned System, JPL, Pasadena, CA, USA, Lesson Number 1196, Jan. 24, 2002. [Online]. Available: https://llis.nasa.gov/lesson/1196
5. G. Szabo. *TAP - Test Anything Protocol*. (2013). Perl Maven. [Online] https://perlmaven.com/tap-test-anything-protocol
6. D. Spinellis, "Notable design patterns for domain-specific languages," *J. Syst. Softw.*, vol. 56, no. 1, pp. 91–99, Feb. 2001, doi: 10.1016/S0164-1212(00)00089-3.
7. D. Spinellis, Z. Kotti, K. Kravvaritis, G. Theodorou, and P. Louridas, "A dataset of enterprise-driven open source software," in *Proc. 17th Int. Conf. Mining Softw. Repositories (MSR)*, New York, NY, USA, Jun. 2020, pp. 533–537, doi: 10.1145/3379597.3387495.
8. D. Spinellis, Z. Kotti, and A. Mockus, "A dataset for GitHub repository deduplication," in *Proc. 17th Int. Conf. Mining Softw. Repositories (MSR)*, New York, NY, USA, Jun. 2020, pp. 523–527, doi: 10.1145/3379597.3387496.