SQLitmus: A Simple and Practical Tool for SQL Database Performance Testing

Aaron Ong

A0122576E

Capstone Initial Thesis for BSc (Honours) in

Mathematical, Computational and Statistical Science

2018

Acknowledgements

1. Assistant Professor Simon Perrault for advising me on writing this Capstone.

2. Professor Olivier Danvy for providing valuable feedback.

3. Hrishi Olickel for advising me to pivot away from my previous topic.

4. Yong Kai Yi for advising me on my user interface.

5. Han Chong for providing me with this much-needed latex template.

6. Ajinkya Chogule for proof-reading this Capstone.

7. Credits to the open source teams from the following projects:

    - React
    - Electron
    - SQLectron
    - React Table
    - React Easy Chart
    - NeDB
    - Sequelize
    - faker

Abstract

This paper presents SQLitmus, a simple and practical tool for SQL database performance testing. SQLitmus was developed to help developers of small-to-mid sized projects conduct quick litmus tests of their SQL databases's performance. With minimal configurations, SQLitmus populates a test database with large volumes of realistic and Schema-compliant test data, and runs randomized queries against the database to analyze its performance. The graphical interface also offers a data plotting and filtering tool to help developers visualize their performance test results.

SQLitmus is compatible with Windows, MacOSX, and Linux machines and supports MySQL, PostgreSQL, and MariaDB databases.

The pilot study was conducted to test SQLitmus against three databases: MySQL, PostgreSQL, and MariaDB. All of these databases are systems provisioned by Amazon Web Service's Relational Database Service (AWS RDS).

The results demonstrates that SQLitmus is capable of generating repeatable and reliable performance analyses of SQL databases. The software recorded clear trends of SQL databases slowing down as their size (amount of data stored) and workload (number of concurrent connections) increased.

Results also revealed performance discrepancies across databases running on identical hardware, data-set, and queries. This shows that SQLitmus can provide developers with intelligence to decide between replaceable databases, queries, and data storage options (e.g., time-stamp vs. date object).

Contents

List of Figures

Chapter 1

Introduction

With the ever-growing volume of internet traffic, and the lasting popularity of SQL databases[1], SQL database performance testing tools are increasingly necessary.

Current SQL database performance analysis tools usually falls under one of the three categories:

- Database benchmarking tools: uses stock schema [2], data, and queries [3] to test for a particular development database's[4] performance.
- Test data generation tools: allows developers to populate a development database with test data[5] that complies with the database's schema. The test data is used to simulate the large load of data that production databases[6] face.
- Live performance monitoring tools: connects to a database in production and

[1] A class of databases that uses a Structured Query Language (SQL) to store and retrieve structured data.
[2] Specifies the types of structured data a SQL databases allowed to store
[3] The mechanism by which SQL databases receives read or write requests
[4] A database used for testing purposes
[5] Meaningful data with statistical properties that approximate a production environment
[6] A database that serves its intended end-users.

monitors all of the transactions carried out by the database.

1.1 A Case for Using Test Data Generation Tools

While test data generation tools are the least used in industry today, this sub-section makes a case for developers and database administrators (henceforth represented by the term: developers) alike to include test data generation tools into their SQL database testing work-flow.

Database benchmarks, while robust, fast, and easy to use, are tested using stock schema and data. Such benchmarks provide developers with a good estimate of their database system's general performance. However, developers are not allowed to configure benchmarking software to use their custom schema, test data, or queries.

Live performance monitoring tools, on the other hand, does not simulate a database's performance. Rather, it pulls performance data from the actual database in production and displays it on a dashboard for database administrators to monitor their database system's actual performance. While they generally provide the most accurate measure of a database system's performance by monitoring actual workloads, they have several shortcomings.

Firstly, Live performance monitoring tools are only able to spot performance issues after they have occurred in a production environment, when fixing performance issues are expensive. Secondly, When fixing performance issues in production, developers are required to backup their data consistently to ensure that the newly deployed fixes do not cause them to lose important data. As such developers are

only able to make conservative fixes after the database is deployed into production.

Test data generation tools are a less reliable, less rigorous, and more cumbersome form of testing. As such, they are often excluded from SQL database testing work-flows. They do, however, allow developers to test for their database system's approximate performance in production by generating a large enough set of test data to simulate a production database's workload. They also allow developers to spot obvious performance issues in advance of deploying the database system into production. This affords developers the flexibility of making drastic changes to their database's overall design during the development phase where the costs of deploying fixes are low.

That being said, there are significant drawbacks to using test data generation tools. They are generally cumbersome to configure, and require a separate tool to run, measure, and visualize their database's performance. It is usually the case that developers require multiple rounds of test data generation, and performance measurements to be conducted before they are able to gain a sense of their development database's performance.

Generated test data, no matter how well configured, are only an approximation of actual data. Thus, while performance analyses conducted by test data generators are much more reliable that database benchmarks, they still do not compare to actual performance monitoring.

The above discussion points towards the need for a test data generation tool that is simple to use and reliable, such that developers will be convinced of the value of including such tools into their SQL database testing work-flow.

1.2 Claims and Contributions

This paper introduces SQLitmus, a simple and practical tool for SQL database performance testing. It is the first open-source software platform that integrates test data generation and population, test query generation and execution, and database performance visualization within a single tool. (Section 3)

The paper also introduces a new method of query templating that advances upon the query templating technique used in QGEN[9] (Sub-sections 2.2 and 3.3).

1.3 Report Outline

This paper first discusses related works in (Section 2), then articulates the features offered by SQLitmus in (Section 3).

With those features articulated, the paper then performs a walk-through of how SQLitmus can be used to test the performance of SQL databases in (Section 4), discusses some useful performance analysis results yielded from SQLitmus in (Section 5), and finally summarizes the paper in (Section 6).

Chapter 2

Related Work

SQLitmus is in essence both a test data generation, and test query generation tool. This section will discuss related works in academia and industry pertaining to test data and query generation.

2.1 Test Data Generation

Since recognizing the need to measure database performance, frameworks and techniques for generating data have been explored substantially across academia and industry.

Significant researches have been conducted on improving the speed of data generation. Gray [3] presented a technique that allows for fast, parallel generation of data in linear time. The limitation with Gray's model is that the number of processors participating in the parallel data generation process is fixed from the start. Rabl [10]

solved this limitation by proposing a technique that allows for the participation of an arbitrary number of processors while not incurring an increased communication overhead.

Techniques for improving the expressiveness of data generators have also been explored. Bruno [2] presented a flexible and easy to use framework for data generation. Their research introduced a graph-based evaluation model which is capable of modelling data distributions with rich intra-row[1] and inter-table [2] correlations. Houkjaer[4] also presented a similar graph-based model but diverges from Bruno's[2] model by achieving intra-column [3] dependencies at the cost of enforcing sequential data generation.

Notable academic works implemented in industry include DBGEN for TPC-H benchmarking [8] and MUDD for TPC-DS benchmarking [12]. Both of these are data generators used for benchmarking purposes and cannot be configured to support custom schema.

While academia have advanced many techniques for generating test data more quickly and expressively, few commercially available data generators developed to this date have implemented these advanced techniques.

Where there are many factors that may have contribute to such an outcome, the author believes that it is simply due to a lack of economic incentive. Test data generation tools are not a part of most SQL database performance analysis workflows, and the economic pressure on companies developing such software can be

[1]Data dependencies across a single row
[2]Data dependencies across a foreign-key relation
[3]Data dependencies down a single column

seen from the price discrepancy between test data generation tools and database monitoring tools.

An additional factor may also be the fact that most test data generation tools generate data using a client-side computer, and the large data-sets they are tasked to generate warrants that the data-sets are stored on-disk as opposed to in-memory. As such, test data generation tools in practice hardly benefit from generating data in parallel, and developers are disincentivized to use test data generation tools due to the long time it takes to run a single test.

Notable commercially-available test data generation tools that support custom schema and referential integrity includes DTM[4], Red Gate[5], and dbForge[6]. All of the above-mentioned software feature a rich set of configurations and data generation logic that allows developers to generate relatively expressive data-sets. They all also feature seeded random data generation, which affords developers the ability to generate identical sets of data across multiple databases. This allows for a fair benchmarking test to be conducted across databases.

Red Gate provides developers with the additional functionality of importing data from existing sources. This allows the test data generation tool to generate its test data from the most reliable source of data - the actual data retrieved from the production database.

DTM, on the other hand, offers developers with the widest range of test data generation functions and expression processors. Developers are able to define com-

[4]http://www.sqledit.com/dg/
[5]https://www.red-gate.com/
[6]https://www.devart.com/dbforge/sql/data-generator/

plex dependencies across rows and down columns using DTM - the most expressive test data generation tool available in the market.

dbForge, while relatively more modest in its offerings, also has a query generation feature.

DTM (Standard edition), Red Gate, and dbForge retail for US$149, US$369, and US$249 respectively.

None of the above-mentioned generators offer parallel data generation or the ability to define data dependencies across relations or tables. They also require a substantial amount of effort to configure, and do not support performance analyses.

2.2 Test Query Generation

On the other hand, Test Query Generation is a much less researched field. Poess[9] presented QGEN, a query generation technique that couples the use of a query template and a query preprocessor. QGEN's preprocessor parses a query template for values marked for substitution and substitutes them using one of the the supported substitution rules:

- Random substitution: returns a normally or uniformly distributed integer within a supplied range.
- Distribution substitution: returns a random element picked from a list of values generated by the MUDD data generator[12].
- Text substitution: returns a random element picked from a list of custom

values and weights.

When QGEN is used to generate queries against a data-set generated from MUDD, the queries yield more accurate benchmarking results - This is due to the fact that the substituted values belong to the same domain as the values generated in the database.

Regardless of the high likelihood of generating queries that target data that actually resides on the database, QGEN still faces two critical flaws:

1. It has no way of ensuring that the generated queries are targeting available data.

2. It has no way of ensuring that the impacts of write queries such as INSERT and DELETE do not modify the cardinality[7] of the data-set.

That being said, QGEN presents a noticeable improvement upon the query generators used in notable data-warehouse benchmarks such as TPC-D, TPC-H, and TPC-R which uses simple substitution mechanisms that are not aware of the data present in the data-set.

2.3 Implications on SQLitmus

SQLitmus began as a project to develop a test data generator which allows developers to specify complex intra-row, intra-column, and inter-table data dependencies. The project aspired to make a highly expressive open-sourced data generator that the developer community could benefit from.

[7]The number of rows of data in the database

However, the author of SQLitmus was compelled by the following reasons to direct the vision of SQLitmus elsewhere:

1. There is already a large pool of expressive test data generation tools available.

2. Test data, no matter how expressive, still relies on the developer to research and approximate production data in their test database. The incremental benefit of developing a more expressive test data generation tool provides marginal improvements to the reliability of its derived performance analysis.

3. A more expressive test data generator will require more time to configure.

4. Most of the test data generation tools available are tailored to the needs of large corporations. The type of test data generation tool that the open-sourced community demands is likely to differ.

The author of SQLitmus identified the following interests of developers using open-sourced tools. Firstly, the tool must be easy to use. Few developers will spend more than a few hours figuring out how to use an open-sourced tool. Secondly, the tool must provide convenience. No developer will use a tool that makes their development process more difficult. Lastly, the tool must be reliable. Tools that are unreliable are generally a waste of time.

Owing to the above-mentioned reasons, SQLitmus pivoted into an open-sourced project to make database performance testing fast, convenient, accessible, and reliable.

2.3.1 Data Generation Implications

The new direction meant that SQLitmus is no longer expected to provide highly expressive data. A trade-off was made among convenience, speed, and expressiveness.[8] Nonetheless, SQLitmus continues to borrow some of the advanced data generation techniques discussed in (Section 2.1) to provide a robust data generator.

To support referential integrity, SQLitmus determines the priority of data generation through a graph model. SQLitmus's graph model is adapted from the graph models proposed by Bruno [2] and Houkjaer [4]. Figure 1 details where the three models converge and diverge.

	[Houkjaer et al, 2006]	[Bruno & Chaudhari, 2005]	SQLitmus
Priority	Determines data generation priority between tables	Determines data generation priority between tables and columns	Determines data generation priority between tables and columns.
Nodes	Nodes carry table data	Nodes carry table data	Nodes carry table data and field data. Nodes carry information on foreign keys.
Edges	Three types of directed edges	One type of directed edge	Three types of directed edge
Edge data	Edges carry information on foreign keys and cardinality distributions	Edges carry no information	Edges carry no information. A seperate data model supplies cardinality distributions.

Figure 2.1: Table of graph models

Priority: Like Bruno's[2] model, SQLitmus's graph model is capable of determining the order of data generation between tables and columns. While SQLitmus's graph model is capable of supporting intra-row data dependencies, the feature has been excluded - since offering intra-row dependencies necessarily clutters the GUI, forces developers to make additional configurations, and slows down the data gen-

[8]Expressive data involves more data generation rules, hence takes longer to generate.

eration process. The graph model is currently used to specify the data generation order between tables, and to delay the generation of self-referential foreign keys. Self-referential foreign keys must necessarily be generated after the column of the same table that it references.

Nodes: Nodes in SQLitmus stores referential constraints on a field level which are then parsed to generate referential constraints on a table level. Storing referential constraints on a field level allows SQLitmus to generate self-referential foreign keys and composite foreign keys with more ease.

Edges: Like Houkjaer's model, SQLitmus's graph model supports Normal, Forward, and Backward edges. Simple and composite primary and foreign keys are supported. Only simple self-referential foreign keys are supported.

2.3.2 Query Generation Implications

On the query generation front, SQLitmus employs a similar methodology as QGEN[9]. It also uses a combination of query templates and a query preprocessor to generate random queries of high quality. In fact, SQLitmus's query generator arguably generates random queries of higher quality as compared to the query templating function employed by QGEN. It requires no additional configurations as it simply uses the exact same set of data generators already configured for use by the test data generator. SQLitmus's templating options are also much more robust and flexible as compared to those available in QGEN. It provides functionality that addresses the two critical flaws of QGEN's query generation technique discussed in (Subsection 2.2). It is able to guarantee that every SELECT, INSERT, UPDATE, and DELETE

query affects at least one row of available data when used correctly. It is also able to reverse the impact of single-row INSERT and DELETE statements and ensure that the cardinality of the data-set does not drift off too much as test queries are being executed. Its only critical flaw is that it is currently unable to reverse the impact of bulk delete statements.

Techniques to ensure zero cardinality drift[9] of the test data-set, while not implemented, have been devised, and will be presented in (Subsection 6.2).

2.3.3 Random Number Generator (RNG)

To generated deterministic random sets of data across multiple test sessions and databases, SQLitmus relies on the PCG-XSH-RR RNG introduced by O'Neill [7]. The above-mentioned RNG belongs to the permuted congruential generator (PCG) family of random number generators. The RNG was selected for the following properties:

- Seeded - Allows for identical RNG sequences to be generated across different runs.
- Fast - Generates the next state at a very low constant cost
- Logarithmic random access - Able to jump ahead to the n^{th} number in the RNG sequence in $log(n)$ time.
- High periodicity - A period of 2^n where n is the number of bits used to store the state.

[9]The phenomenon where the tested queries modify the cardinality of the data-set through INSERT or DELETE statements. If unaccounted for, results in unreliable performance analyses.

- High uniformity - At any point of the random number generation process, generated numbers occur at highly uniform frequencies across the range.

SQLitmus employs a 64-bit implementation of the RNG which is has a period of 2^{64}, much higher than the current data size generation limit that SQLitmus supports. The fast logarithmic random access times when used in conjunction with the PDGF seeding strategy proposed by [10] and [1] will enable SQLitmus to generate data with intra-row, inter-column, and inter-table dependencies without performing expensive disk reads. This feature while impressive, requires a fair bit of effort to implement, and is thus de-priotized to a later patch.

SQLitmus also utilizes a good multiplier constant proposed by [6] to ensure a high uniformity in randomly generated numbers. A poorly chosen multiplier constant will also reduce the period of the RNG.

2.4 Trade Offs

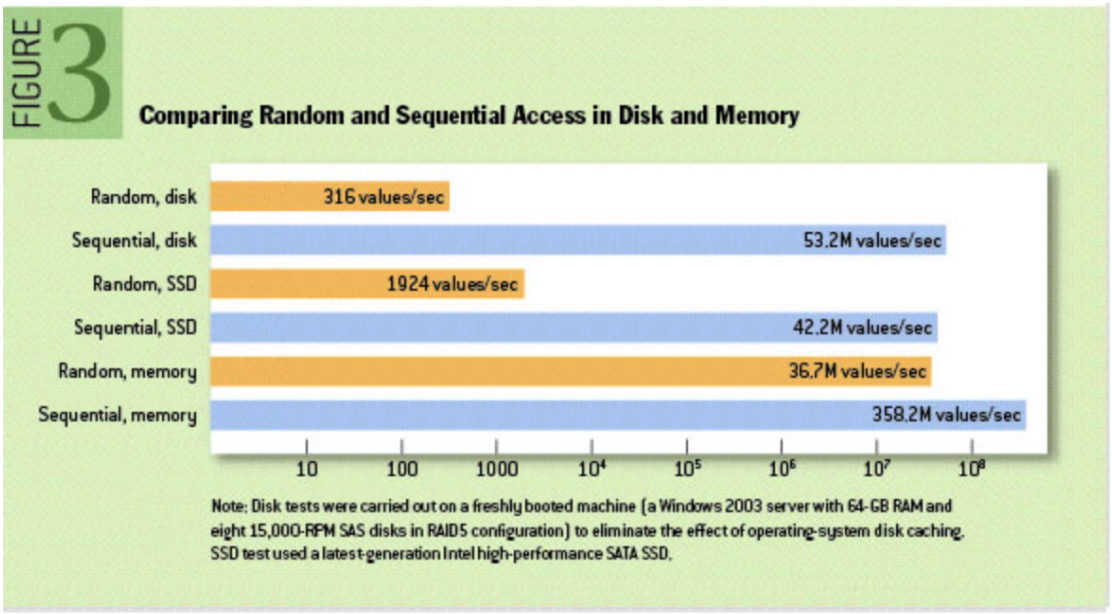


Figure 2.2: Performance of RAM, SSD, and HDD [5]

For each test, SQLitmus is only able to generate data up to the the amount of RAM available on the client-side computer. This is because SQLitmus stores the entire data-set in memory. This trade-off is made for the following reasons:

- While SQLitmus generates and is capable of storing data sequentially, the database currently used to manage the data generation process, NeDB, only allows random access updates.
- Data generation will be approximately 1,000,000 times faster on RAM vs. HDD
- Data generation will be approximately 200,000 times faster on RAM vs. SSD

- There is no need to persist the data generated by SQLitmus (It is already on a database)
- Developers of small-to-medium sized applications rarely store more than 10GB of data
- Large binary objects such as flat files and pictures are usually stored in the server.

To reiterate, the key goal of SQLitmus is to to make database performance testing fast, convenient, accessible, and reliable. Initial trade-offs were made in favour of speed as opposed to the maximal data-set size. As the author believes that an approximate 10 GB limit is sufficient for most developers using the tool. SQLitmus currently takes approximately a minute to generate 1,000,000 fields of data. Thus, storing the data-set on-disk at this moment would be untenable.

Nonetheless, SQLitmus is intending to rework its data generation process to off-load most of the in-memory data storage to disk (Since transferring a random access load in-memory to a sequential load on-disk actually provides a performance benefit). If this effort is successful, SQLitmus will be able to generate data at an almost constant speed while requiring that only a fraction of the data-set currently being generated resides in memory. This future work will be discussed in (Section 6.2).

Chapter 3

SQLitmus Features

Every good software begins by serving the needs of its target users well. This section begins by enunciating the user stories that developers of small-to-medium sized applications want from SQL database performance analysis tools.

As a user, I want...

- A performance test that is reliable, so that I am sure that the performance testing is accurate.
- A performance test that is repeatable, so that I am able to test for the impacts of the new configurations I made.
- A performance test that is repeatable, so that I am able to select the most suitable database system to use.
- A way to save my configurations, so that I do not have to waste time reconfiguring my tests.

- An easy and convenient way of testing my SQL databases, so that I can focus my efforts on developing my software.
- A good way of visualizing my performance testing data, so that I do not have to waste time exporting them into another software.
- A way to test my database's performance under multiple configurations, so that I can identify performance bottlenecks and trends more effectively.
- A way to test my database under different data loads, so that I have a sense of how well my database scales with an increasing data load.
- A way to test my database under different numbers of concurrent connections, so that I have a sense of how well my database scales with an increasing number of concurrent users.
- A way to test my database with multiple types of queries, so that I have a more complete understanding of my database's performance.
- A way to specify the types of data I wish to generate, so that I can ensure that they are compliant with my database schema.
- A way to specify the types of queries I wish to test, so that the test results are more representative of my software's actual performance.
- A software that guides me through the configuration process, so that I do not have to spend time figuring it out.
- A way to test my database quickly, so that I do not have to wait too long to see the results of my test.

With the user stories enunciated, the remainder of this section demonstrates how SQLitmus's feature set provides a solution for all of the above-mentioned user stories.

Note that the feature set in this section is often explained through the use of the test scenario presented in (Section 4). While this section is designed to be self-sufficient, readers may opt to refer to the test design in (Section 4) to gain a clearer understanding of the specific scenario used. For readers who are unfamiliar with how SQL databases operate, the author recommends reading (Section 4) first.

3.1 Database Connection Management

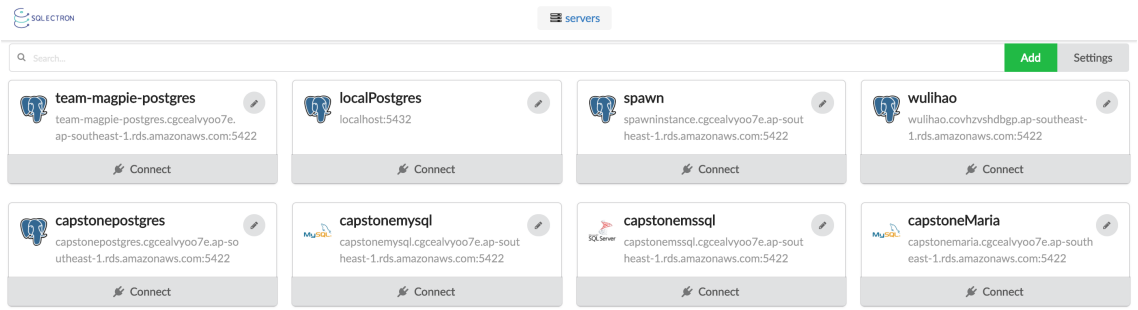


Figure 3.1: SQLitmus landing page

SQLitmus's landing page (see Figure 3) offers a way for developers to manage and persist their database connection configurations. Developers are able to add a new set of connection settings, or update their existing set of connection settings. The database management dashboard was forked off the open-source tool SQLectron which already provided a user-friendly GUI for database connection management. Beyond persisting database connection settings, SQLectron does not support any further types of persistence.

To afford developers an additional layer of convenience, SQLitmus persists all available configurations discussed in later sections on a database level. This prevents configurations made for a particular database to spillover into other databases even when they belong to the same server.

3.2 Data Generation

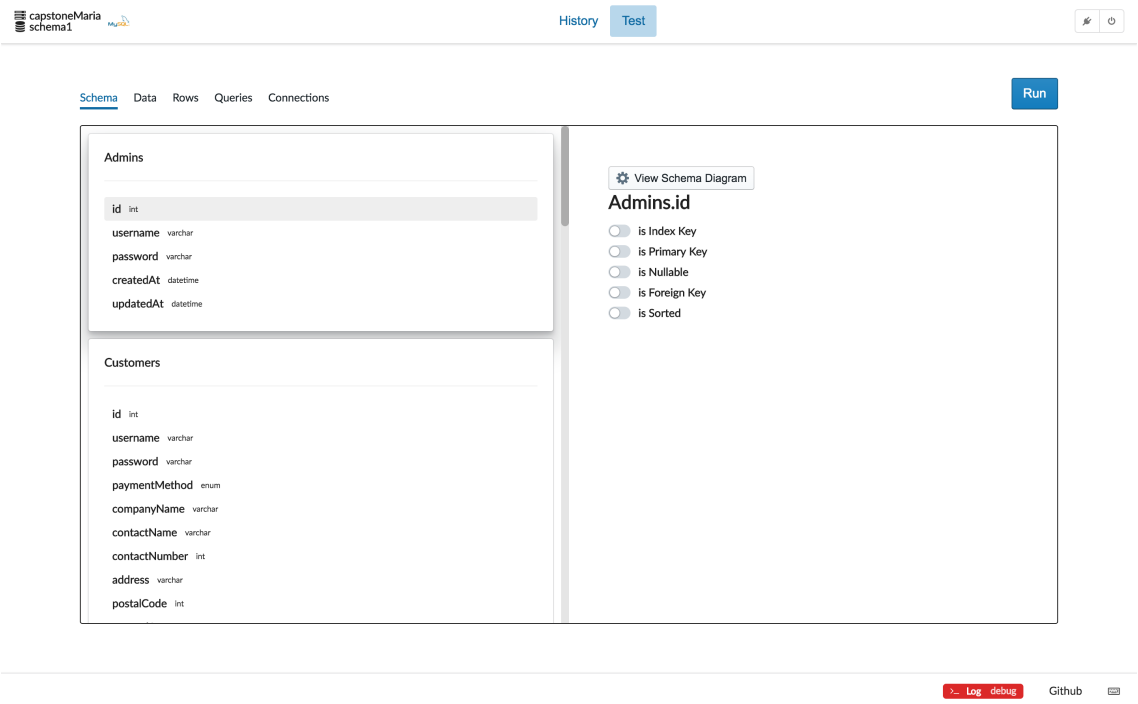


Figure 3.2: SQLitmus automatically populating tables and fields

Upon connecting to a specified database, SQLitmus populates the the GUI with the full list of tables and fields present on the database (excluding system databases). Field types are also auto-populated.

Figure 4 displays how the SQLitmus GUI looks like upon connecting to a database

for the first time. Without any configuration, relevant details of a database's schema is populated by default into SQLitmus.

3.2.1 Configuring Field Constraints

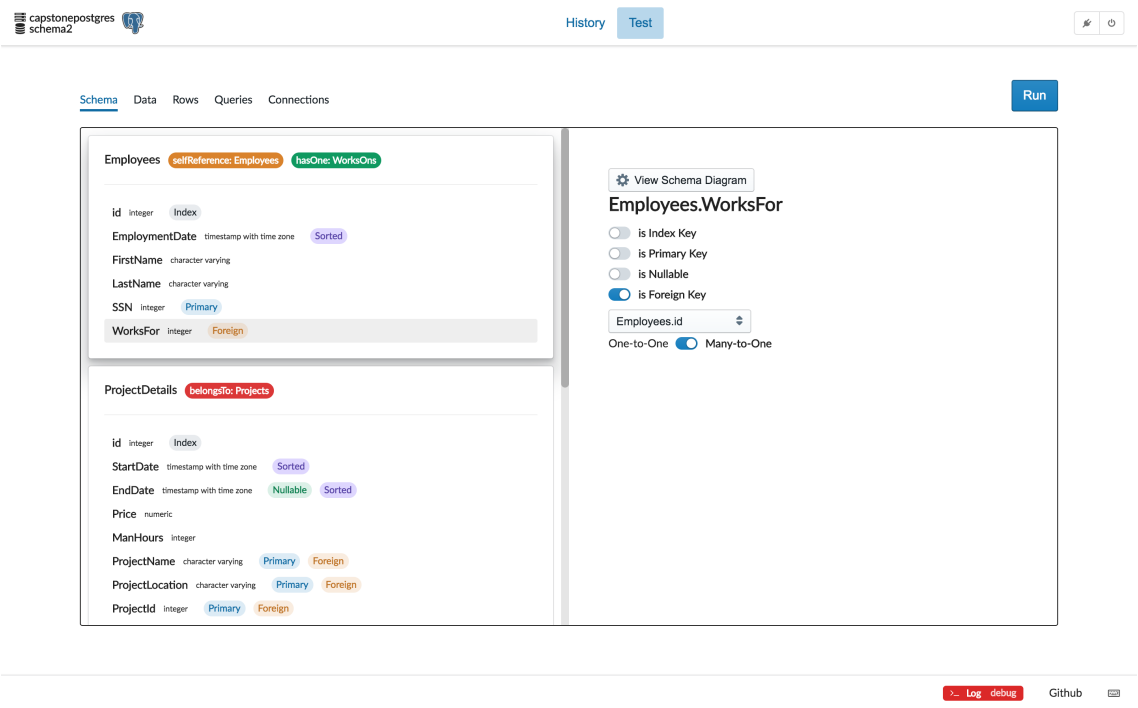


Figure 3.3: Field constraint configurations

SQLitmus supports six different forms of field constraints[1]: Index Key[2], Primary Key[3], Nullable[4], Foreign Key[5], Unique Key[6], and Sorted[7] constraints.

As users configure field constraints, the GUI provides clear visual feedback. This is demonstrated in Figure 5, where field constraints appear as tags next to their

[1]Specifies the rules that fields in a SQL database has to comply with
[2]An integer field that is guaranteed to be unique down a column
[3]A single or combination of multiple fields that guarantees a record's uniqueness
[4]A field that is allowed to hold null values
[5]A single or combination of multiple fields that guarantees a unique record is referenced.
[6]A field that is guaranteed to be unique down a column
[7]A field that is guaranteed to be sorted down a column

relevant fields and foreign key constraints appear as tags next to their relevant tables.

Developers are able to specify their field constraints though simple switches and selections. No text or numerical inputs are offered. This achieves a twofold purpose: simplify the configuration process, and eliminate the risk of misconfiguration altogether.

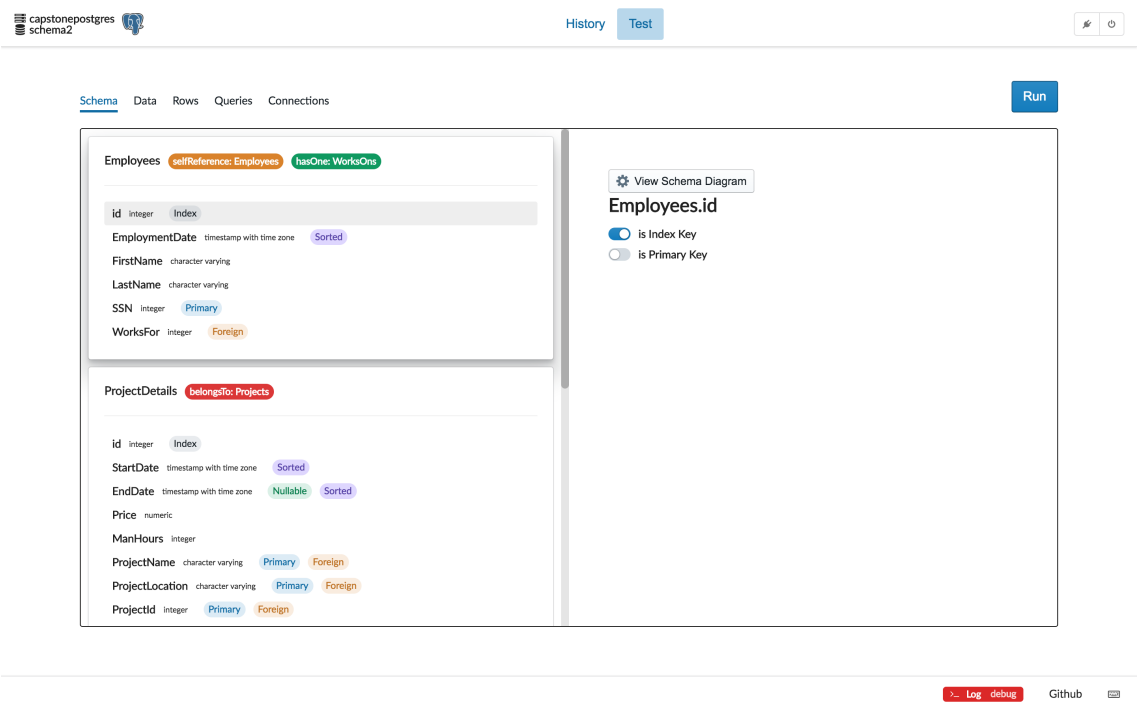


Figure 3.4: Selecting upstream constraints disable invalid downstream configurations

As developers specify upstream constraints, invalid downstream configurations are disabled.

This behavior is observed in figure 6 where configuring an index field disables developers from specifying other invalid downstream configurations. The rationale is as follows - index fields necessarily cannot be foreign keys, or nullable fields. It is

also taken for granted that index fields are unique and sorted.

SQLitmus also limits developers from specifying configurations that are disallowed by databases (e.g., specifying an index key on a non-integer field), considered anti-patterns in the developer community (e.g., using timestamps as primary or foreign keys), or breaks referential integrity (e.g., selecting a non index, primary, or unique key as a foreign key target).

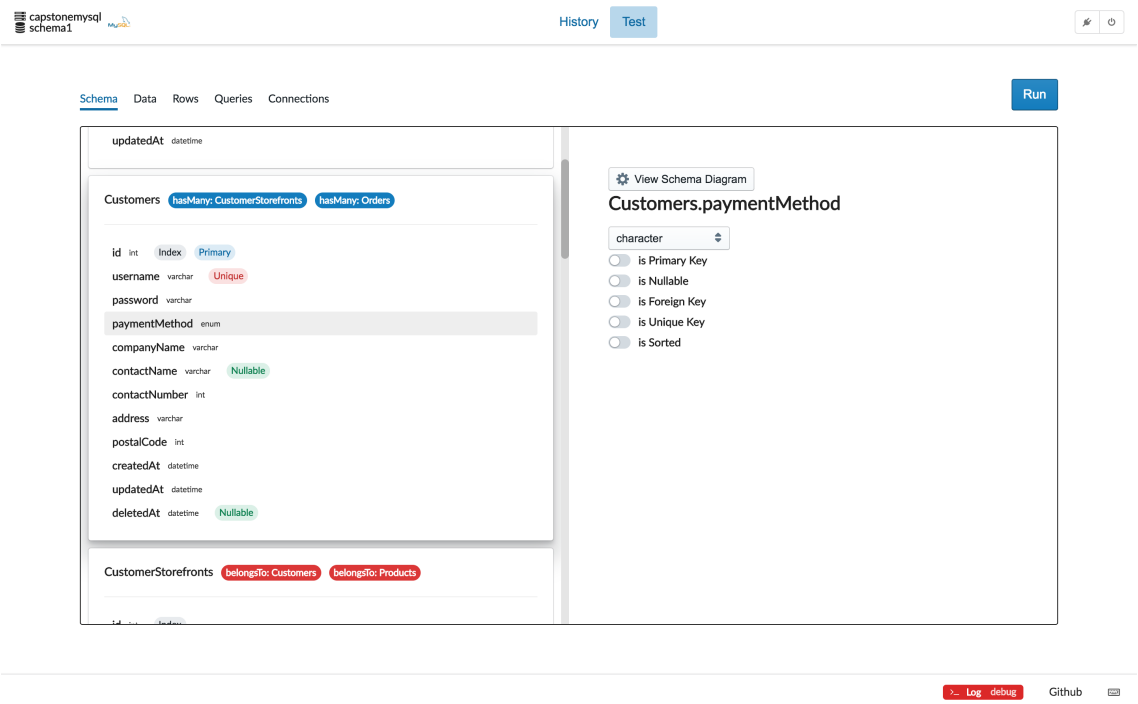


Figure 3.5: Unsupported data type configuration

For unsupported data types, SQLitmus allows developers to configure the field type according to one of the six supported types: integer, character, numeric, boolean, time, json. For all downstream configurations, SQLitmus will treat the unsupported type as the type configured by the developer. Figure 7 demonstrates how unsupported data types can be configured into one of SQLitmus's supported types.

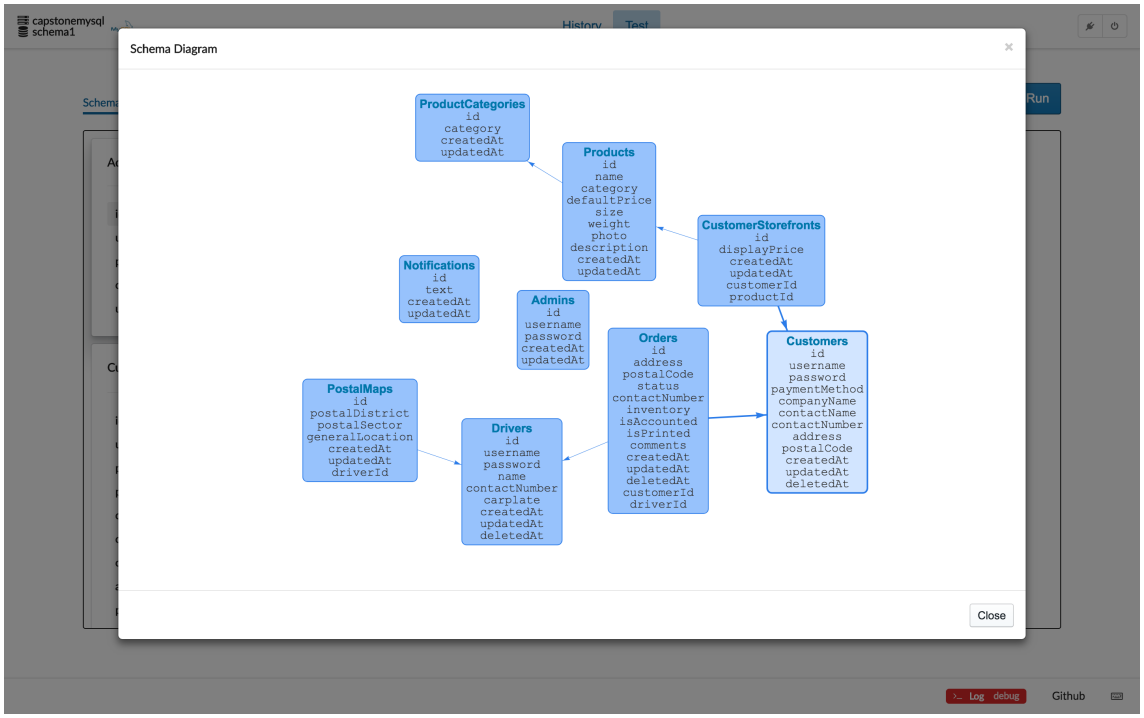


Figure 3.6: Visualization of schema diagram

Developers are also able to view a diagram of their schema to quickly identify misconfigured foreign key relations. Figure 8 demonstrates one such diagram.

3.2.2 Configuring Data Generators

While data generators can be configured at any point of the configuration process, the author recommensd that data generator configurations are performed after field constraints configurations. This allows for a smoother configuration process as some data generators may be invalidated by changes in field constraints.

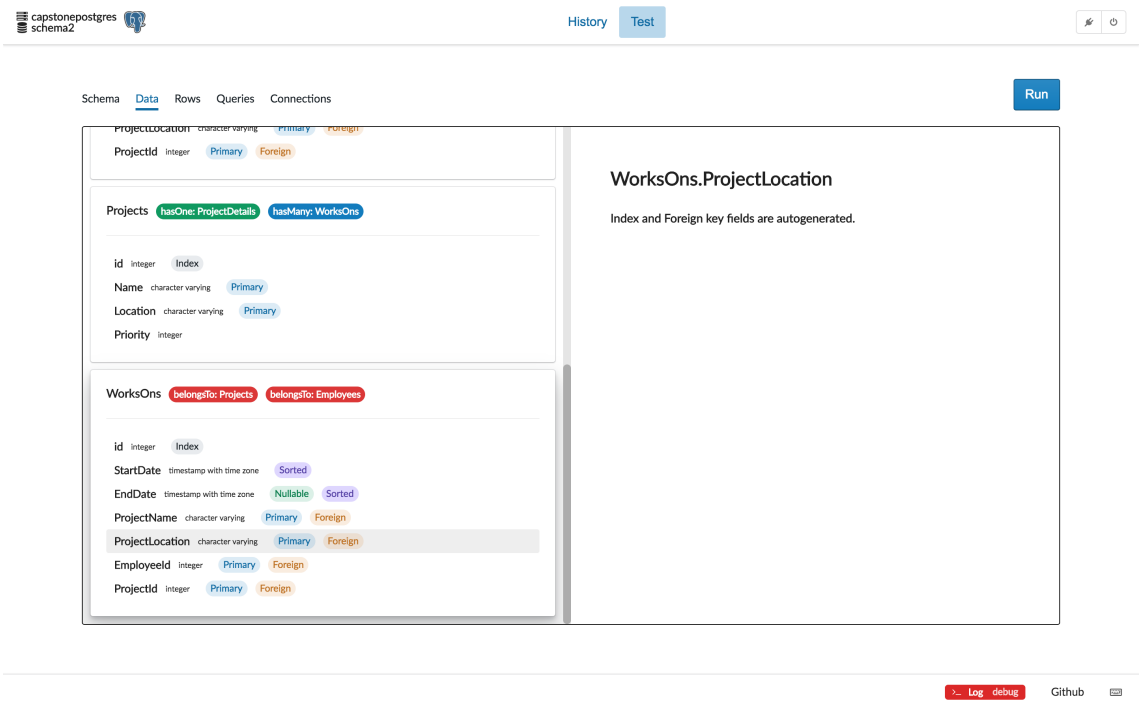


Figure 3.7: Automatically generated primary and foreign keys

In SQLitmus, index and foreign key fields are automatically generated. It is complex to resolve referential constraints, especially for cases where a composite primary key is composed of multiple simple and composite foreign keys (WorkOns table in Figure 9). Thus, such concerns are abstracted away from developers.

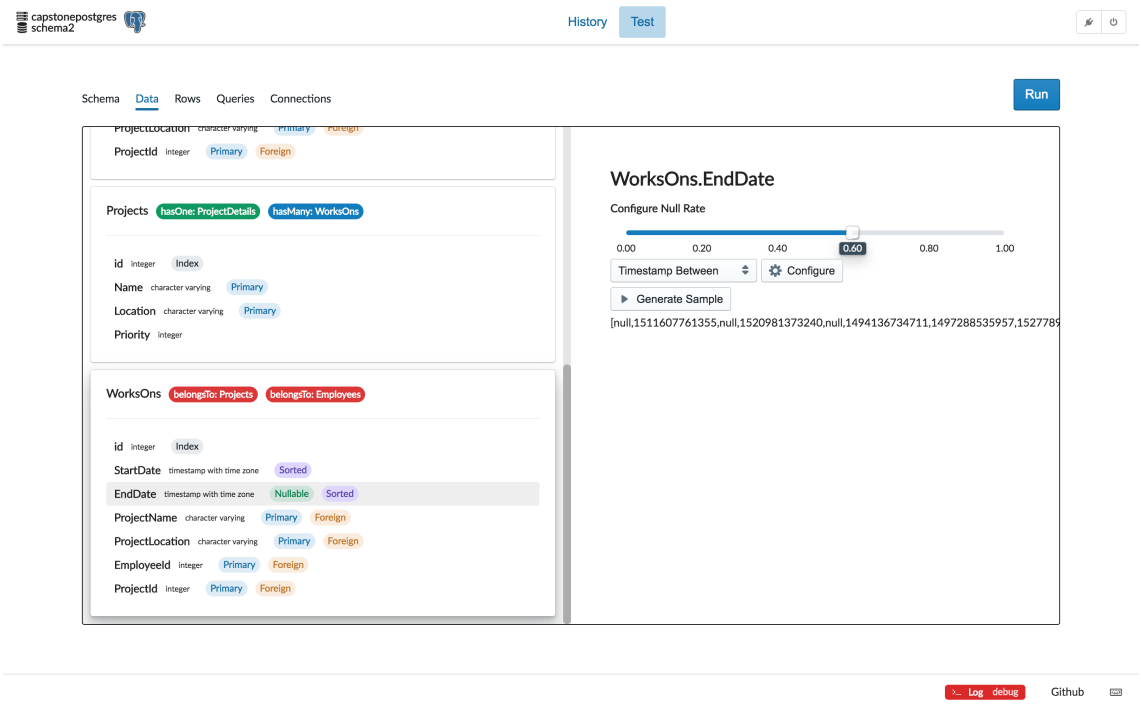


Figure 3.8: Configuring null rates

In the case where a nullable field is specified, SQLitmus offers an additional null rate generator that is compatible with all of SQLitmus's 23 built-in and 5 custom data generators. Developers simply has to specify a null rate on top of their chosen data generator. Figure 10 displays the slider in SQLitmus's GUI that configures null rates.

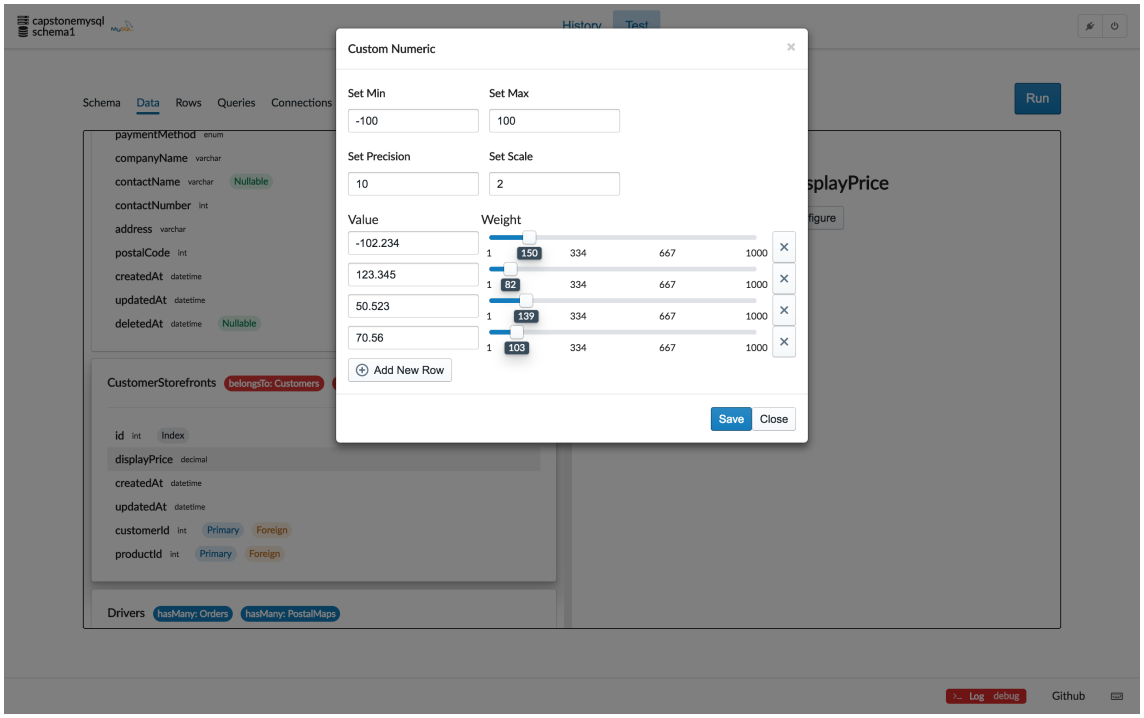


Figure 3.9: Data generator configurations

Each data generator comes with its own set of configurations which developers are able to access through clicking on the configure button next to the data generator. Some validation rules are common to all generators of a given type (e.g., Numerical generators all allow developers to specify min, max, precision[8], and scale[9] - seen in Figure 11). SQLitmus also ensures that improper validation rules do not affect the integrity of the generated data [10]. These validation rules allow developers to generate data that is compatible with the most stringent database settings.

[8]The number of digits in a number.
[9]The number of digits after the decimal point.
[10]e.g., positive scales for integer fields are always considered to be zero

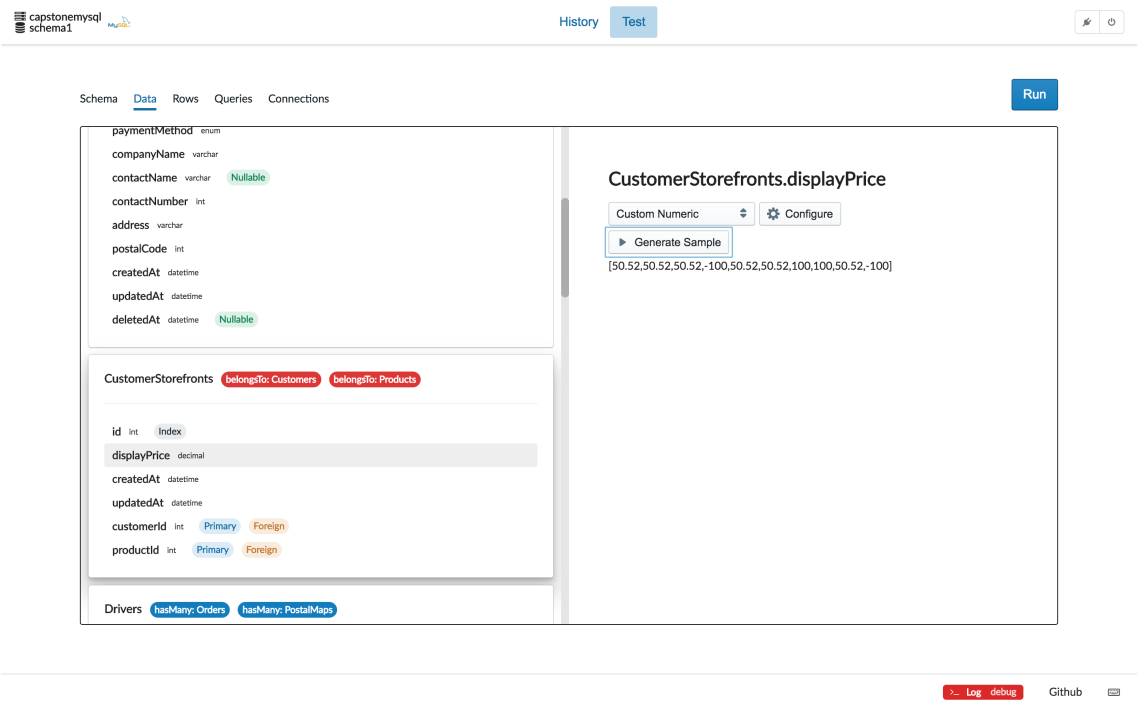


Figure 3.10: Sampling data from configured generators

Upon successful configuration, developers are able to generate data samples from their customized generator to ensure specification adherence. Figure 12 demonstrates the generation of data samples from a configured generator.

3.3 Query Generation

Whereas data generation sets the database up with a valid dataset to prepare it for performance testing, query execution is where performance testing actually occurs. SQLitmus employs the most objective method of performance analysis available: measuring and recording a query's response time.

The query generation feature presented by SQLitmus substantially improves the

developer's user experience in configuring test queries. Instead of requiring that developers input an exhaustive list of valid queries, SQLitmus allows developers to specify the types of queries they wish to test their database with. SQLitmus then generates hundreds of randomized valid queries to test the database with for each specified query type. To achieve this end, SQLitmus offers query templating, and a GUI for developers to test the validity of their query templates.

The value of query generation is as follows:

- It is far less rigourous for developers to design a query template as opposed to designing a large set of possible queries to test their database with.
- Performance analysis using a static list of queries yields low quality performance data. Databases cache query results, thus identical queries when ran multiple times against the same database, will measure the speed the database takes to deliver a pre-loaded response from its cache rather than the actual time a database takes to process the query.
- Since the set of generators used to populate the database is identical to the set of generators used to populate the queries, developers are guaranteed to generate valid queries and benefit from the high likelihood of their queries targetting actual data in the database.

3.3.1 Query Templating

Developers are often more interested in finding out the performance of a type of query as compared to the performance of an actual specific query. Developers also do not wish to relearn a new Domain Specific Language to specify their types of

queries, and wish to test the exact queries executed in production.

Query templates afford developers the ability to generate multiple queries from a single template .

```
-- WorksFor targets the index of another row in the Employees table and
-- specifies a supervisor-subordinate relationship
SELECT * FROM Employees WHERE WorksFor=${Employees.RANDROW};
```

For instance, the above code snippet demonstrates how a simple query template in SQLitmus allows developers to test for their database's ability to retrieve all subordinates of a randomized Employee. Compare this to having the developers specify a static list of queries.

```
SELECT * FROM Employees WHERE WorksFor=1;
SELECT * FROM Employees WHERE WorksFor=2304;
SELECT * FROM Employees WHERE WorksFor=3520392;
```

Not only is templating much more convenient, it also has a low learning curve. The template above also guarantees that the substituted value belongs to the same domain as the data generated in the WorksFor column.

Expressions nested between a dollar sign and curly braces `${expression}` are parsed by the query preprocessor and replaced by an appropriate value.

In the above case, SQLitmus will replace the expression with a randomly generated row index from the Employees table.

Beyond generating substitution rules, SQLitmus's query templates affords developers with the ability to setup their queries (to ensure that an actual workload is

being measured) and perform clean-ups (to ensure that INSERT or DELETE queries do not cause a substantial cardinality drift) through the use of special delimiters.

```
-- Insert new employee (MySQL)
-- Values are (id, EmploymentDate, FirstName, LastName, SSN, WorksFor)
-- SSN is the primary key
-- Only queries specified between the begin and end delimiters will
-- have their response time measured.
DELETE FROM Employees WHERE SSN = ${Employees.SSN};

${BEGIN.DELIMITER}

INSERT INTO Employees VALUES
(null,FROM_UNIXTIME(CEIL(${Employees.EmploymentDate}/1000)),
${Employees.FirstName},${Employees.LastName},
${Employees.SSN}, ${Employees.RANDROW});

${END.DELIMITER}

DELETE FROM Employees WHERE SSN = ${Employees.SSN};
```

For instance, if a developer is interested in testing the time taken to INSERT a new employee record into his/her database, the query response time measurement should ideally reflect the time it takes to INSERT an actual row of employee data into the database.

SQLitmus's query template allows developers to first DELETE any conflicting employee record/records from the database before testing the time it takes for the database to INSERT an actual row of employee data. This functionality allows SQLitmus to yield a more accurate query response time measurement as compared to QGEN.

The impact of the INSERT workload on the cardinality drift of the Employees table can also be mitigated through specifying an additional DELETE statement af-

ter the INSERT is performed. Cardinality drifts are thus much more well accounted for in SQLitmus as compared to QGEN.

Template	Values
${ `TableName` . `FieldName` }	A randomly generated value from any previously configured data generators. Note that this does not work for index and foreign key fields since they are not user-configured.
${ `TableName` .NUMROWS}	The number of rows generated for a specified Table at the current test. Defaults to 10 when used in GUI testing.
${ `TableName` .RANDROW}	A randomly selected value in the range of [1,NUMROWS].
${BEGIN.DELIMITER}	Specifies when to start measuring the query response time.
${END.DELIMITER}	Specifies when to stop measuring the query response time.

Figure 3.11: Full set of Query templating options available in SQLitmus

Figure 13 displays the full set of Query Templating options available in SQLitmus. This combination of templating options allows developers to populate their query templates with random values that belong to the same domain as any given field of interest.

For index fields, developers can use the `${TableName.RANDROW}` templating option to generate a random index of the same domain as the generated data-set.

For non-index and foreign key fields, the `${TableName.FieldName}` option is available.

All foreign key fields necessarily references a root non-foreign key target. For instance, if Table1.Field1 references Table2.Field2, developers can simply use the `${Table2.Field2}` generator to generate values for Table1.Field1. [11]

The following scenario demonstrates how the various query templating options

[11] Use `${Table2.RANDROW}` instead if Field2 is an index key

can be used together to insert a new row of employee data of the same domain as the generated data-set.

```
1   -- Staffing an employee on a project (PostgreSQL)
2   -- (id, StartDate, EndDate, ProjectName, ProjectLocation, EmployeeId, ProjectId)
3   -- ProjectName, ProjectLocation, and ProjectId addresses the
4   -- Name, Location, and id fields of the Projects table respectively.
5   -- EmployeeId addresses the id field of the Employees table.
6   INSERT INTO "WorksOns" VALUES (DEFAULT,
7   to_timestamp(CEIL(${WorksOns.StartDate}/1000)),
8   to_timestamp(CEIL(${WorksOns.EndDate}/1000)),
9   ${Projects.Name},${Projects.Location},
10  ${Employees.RANDROW},${Projects.RANDROW});
```

- StartDate and EndDate has directly accessible generators `${WorksOns.StartDate}` and `${WorksOns.EndDate}` so we supply them accordingly.
- ProjectName and ProjectLocation references non-index fields Projects.Name and Projects.Location so we supply `${Projects.Name}` and `${Projects.Location}` accordingly.
- EmployeeId and ProjectId references the indexes of the Employees and Projects tables so we supply `${Employees.RANDROW}` and `${Projects.RANDROW}` accordingly.

All queries generated are deteministically random, so developers are able to execute the same set of randomly generated queries across different databases and different trials of the same test.

3.4 Query Template GUI

Regardless of how simple it is to use the query templating system, developers need a way of testing the validity of their query templates ahead of running any performance analyses. Developers also need a way to persist the list of already configured query templates in the software so that they do not have to spend time re-configuring the list of queries templates that they have used in a previous test.

Similar to the database connection management module, the query template GUI has also been forked from SQLectron. The SQLectron query browser natively supports tabs, query execution, and query auto-completion.

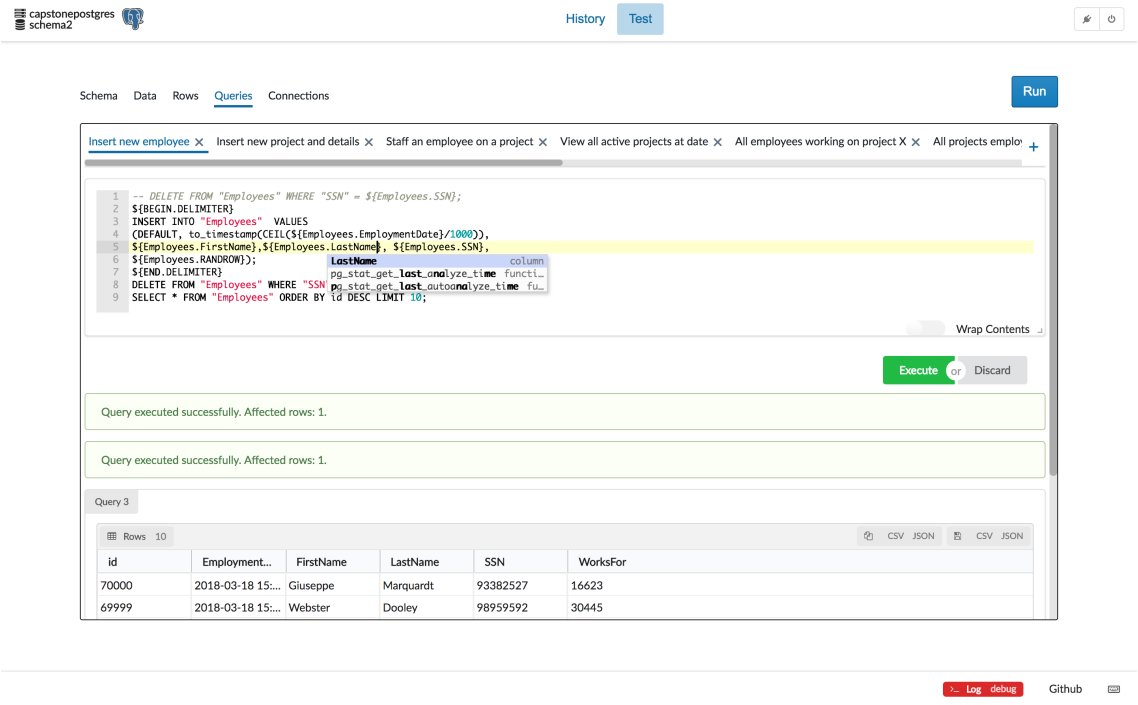


Figure 3.12: Query Template GUI

SQLitmus extends the query browser of SQLectron by offering tab persistence,

query template preprocessing, autocompletion of SQLitmus templating options, and a modified GUI that fits SQLitmus's overall design better. While the GUI (as shown in Figure 14) is densely packed, it still offers a good user experience for developers working on the SQLitmus platform.

Developers are able to test all available SQLitmus templating options within the GUI itself. Without any additional configurations, the query template preprocessor populates the query templates with values generated from the already configured data generators. All test queries executed also target the same database that the developer is currently connected to.

3.5 Test Configurations

Developers do not simply wish to test the performance of their SQL database in a single environment. They wish to test their database under multiple environments so that they are able to identify trends that are important to them such as:

- How well their SQL database's performance scales with the amount of data it holds.
- How well their SQL database's performance scales with the number of concurrent requests it serves.

The following test configurations allow developers to specify up to 25 different environments to test their SQL database under in a single run.

3.5.1 Row Configurations

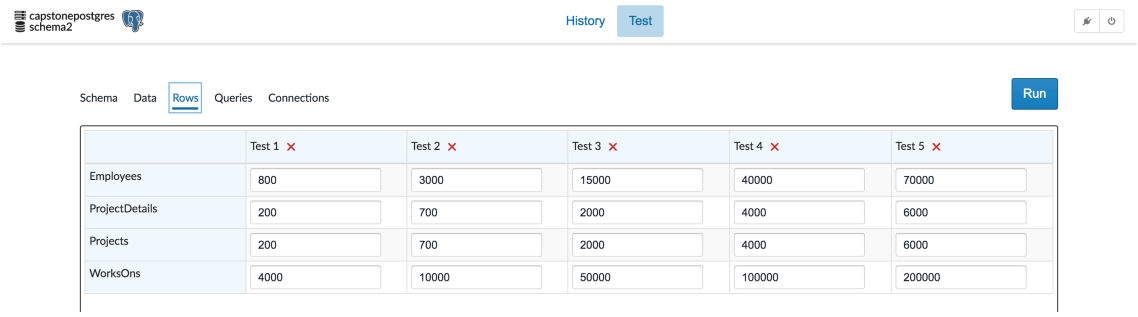


Figure 3.13: Row Configuration GUI

The row configurations tab (as shown in Figure 15) allows developers to specify the number of rows of data they wish to generate on a tabular level. Developers are allowed to specify up to five test trials in a single run of performance analysis.

As the data generated in the last specified test persists in the database, developers who are not concerned with SQL performance testing can still use SQLitmus as a pure test data generation tool. This allows them to re-purpose SQLitmus for API performance testing, and user interface development testing.

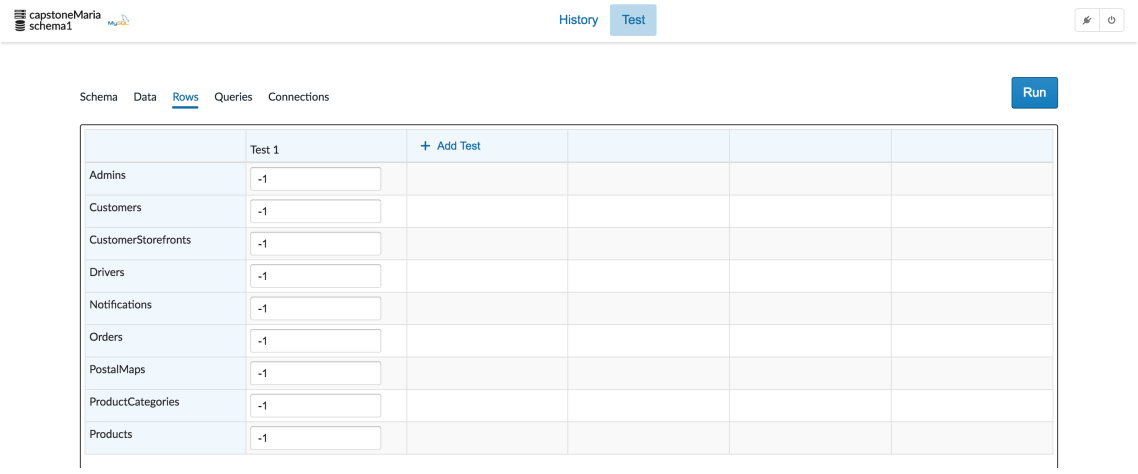


Figure 3.14: Bypassing data generation through invalid row configurations

The author of SQLitmus also recognizes that SQLitmus is neither the fastest, most robust, nor most expressive test data generator available in the market. As such, SQLitmus provides a way for developers using other test data generation tools to bypass SQLitmus's data generation process to solely capitalize on SQLitmus's query templating and performance testing engine to measure their database's performance. Figure 16 shows an example of bypassing data generation in SQLitmus.

If at least one input is configured with an invalid number of rows - being any negative integer - SQLitmus will skip the data generation process and proceed directly to executing the performance test.

If all rows are supplied with zero, SQLitmus wipes the database clean of any test data and resets all automatically incrementing counters present on the database to their default values.

3.5.2 Max Connection Pool Configurations

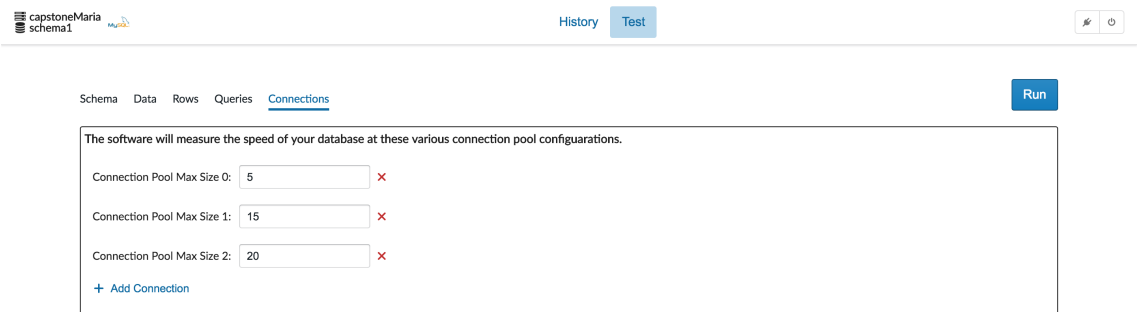


Figure 3.15: Max connection pool GUI

The max connection pool size simulates the ability for the database to handle multiple concurrent requests from a single server. Developers can specify up to five max

connection pool configurations to test their SQL databases at. Figure 17 demonstrates SQLitmus's max connection pool configurations GUI.

3.6 Running a Test

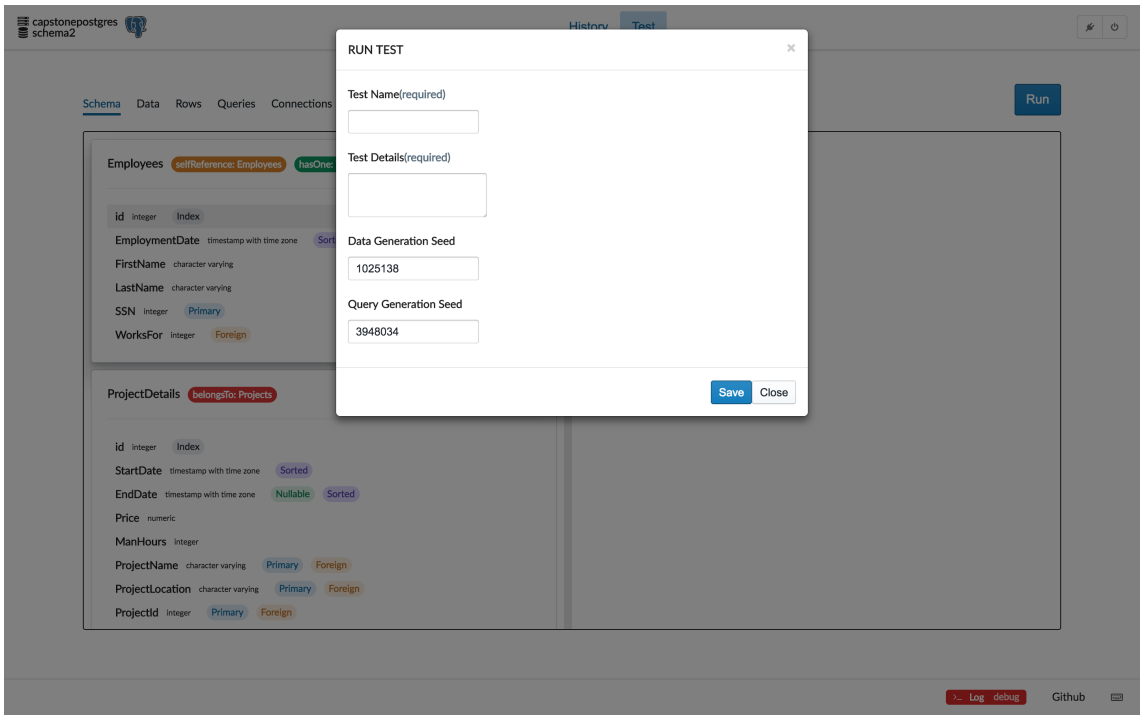


Figure 3.16: Test running modal and its available parameters

After the configurations are complete, SQLitmus allows developers to specify separate Data and Query Generation seeds. Each seed deterministically generates the type of data and queries used in the test. This affords developers the option to test multiple databases, or repeat the same test on the same database using the exact same set of data and queries. The seeds are also separated so that developers are able to generate the exact same set of data but test the data-set using a different set of randomly generated queries to account for the chance that the randomly selected

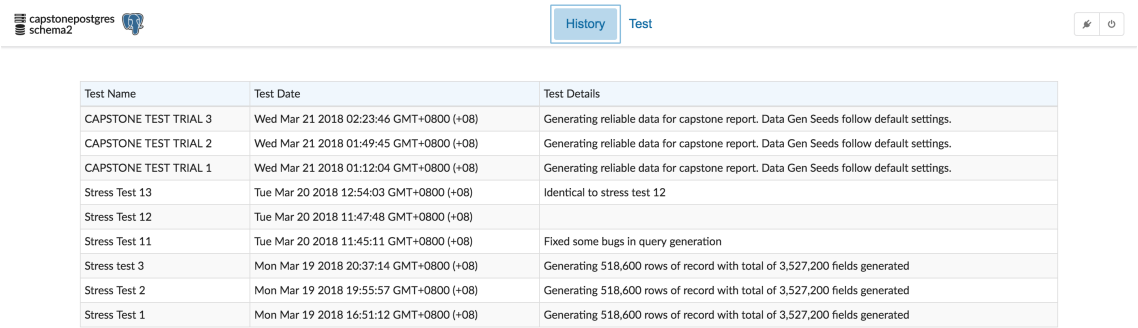
queries were "lucky".

Figure 18 displays the how test parameters can be configured on the test running modal.

3.7 Data Management

After all the performance testing is complete, developers still require a simple and effective way of managing their performance analysis results. While the data can simply be exported to other tools for data visualization purposes, the author of SQLitmus believes that a full feature data management solution optimized for visualizing query response time trends should still be made available within SQLitmus itself. This provides an additional layer of convenience for developers.

3.7.1 History of Test Records



Test Name	Test Date	Test Details
CAPSTONE TEST TRIAL 3	Wed Mar 21 2018 02:23:46 GMT+0800 (+08)	Generating reliable data for capstone report. Data Gen Seeds follow default settings.
CAPSTONE TEST TRIAL 2	Wed Mar 21 2018 01:49:45 GMT+0800 (+08)	Generating reliable data for capstone report. Data Gen Seeds follow default settings.
CAPSTONE TEST TRIAL 1	Wed Mar 21 2018 01:12:04 GMT+0800 (+08)	Generating reliable data for capstone report. Data Gen Seeds follow default settings.
Stress Test 13	Tue Mar 20 2018 12:54:03 GMT+0800 (+08)	Identical to stress test 12
Stress Test 12	Tue Mar 20 2018 11:47:48 GMT+0800 (+08)	
Stress Test 11	Tue Mar 20 2018 11:45:11 GMT+0800 (+08)	Fixed some bugs in query generation
Stress test 3	Mon Mar 19 2018 20:37:14 GMT+0800 (+08)	Generating 518,600 rows of record with total of 3,527,200 fields generated
Stress Test 2	Mon Mar 19 2018 19:55:57 GMT+0800 (+08)	Generating 518,600 rows of record with total of 3,527,200 fields generated
Stress Test 1	Mon Mar 19 2018 16:51:12 GMT+0800 (+08)	Generating 518,600 rows of record with total of 3,527,200 fields generated

Figure 3.17: History of test records

SQLitmus stores a history of all performance analysis records that users have conducted on a database level. The history is sorted in reverse chronological order as

seen in Figure 19 above.

3.7.2 Data Visualization

Figure 3.18: Data visualization module

The data visualization component allows developers to identify trends of their database's performance quickly without having to clean or process their data.

Developers are able to investigate their data with a high level of flexibility as the data visualization component allows developers to choose any recorded numerical data for its x-axis and group the data-points using any recorded numerical or string data. Figure 20 demonstrates how the data visualization module can be used to investigate trends.

The color palette was selected using Hue Separation Lightness (HSL) values to ensure that they are visibly distinct from one another. The graph axes were selected to be the minimum range capable of fitting all recorded values.

The graph and legend was developed through using the react-easy-graph library and the colour selection algorithm was taken from Stack Overflow [12].

3.7.3 Data Filtering

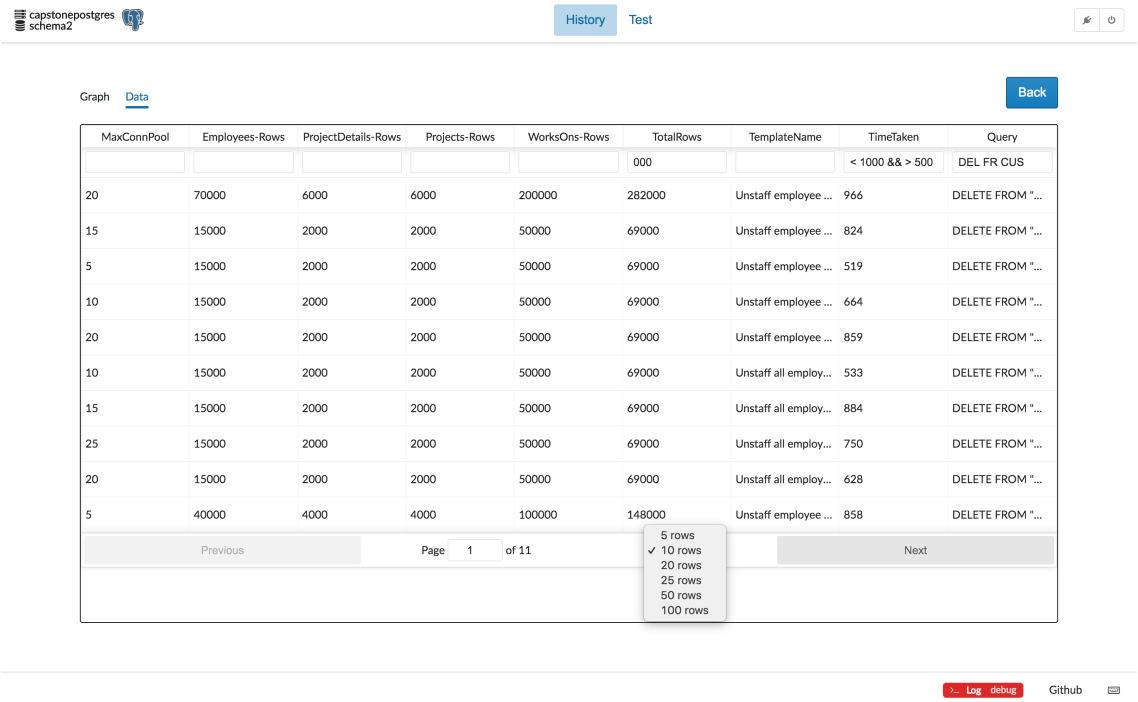


Figure 3.19: Data filtering module

The data filtering component (displayed in Figure 21) provides a way for developers to filter down the the large data-set that they are working with to dig deep into trends of interest.

[12] https://stackoverflow.com/questions/10014271/generate-random-color-distinguishable-to-humans

It allows developers to filter the graphed data-set using a combination of filtering rules specified at a column level. It also uses a powerful filtering engine to afford developers more flexibility.

The word filtering mechanism allows developers to search the string data-sets using intuitive search values. It does not require developers to use complex reg-ex expressions but provides an almost identical level of expressive capability for this intended purpose. It was developed using the help of the match-sorter[13] library.

The number filtering mechanism allows developers to use a combination of intuitive search rules to find their data. The operators supported are:

$$>=, <=, >, <, =, \&\&, ||$$

[13]https://github.com/kentcdodds/match-sorter

Chapter 4

Designing a Robust Test for SQLitmus

To demonstrate the capabilities of SQLitmus, a robust pilot study was devised.

4.1 Test Schema

Figure 4.1: Test Schema for pilot study

The selected test schema was adapted from a similar study conducted by Houkjaer [4].

The schema proposed by Houkjaer tests SQLitmus for the following capabilities:

- Generating simple primary keys (Employees.id)
- Generating composite primary keys (Projects.Name & Projects.Location)
- Generating self-referential foreign keys (Employees.WorksFor → Employees.id)
- Generating simple many-to-one foreign keys (WorksOns.EmployeeId → Employees.id)
- Generating composite one-to-one foreign keys (ProjectDetails → Projects)

- Generating composite many-to-one foreign keys (WorksOns → Projects)
- Generating many-to-many relationships (Employees & Projects through WorksOns)

While Houkjaer's schema allowed us to conduct a robust pilot study on SQLitmus, minor modifications were made to Houkjaer's schema for various reasons. Figure 23 below detail the modifications made to Houkjaer's schema in our pilot study and their related rationale:

Modification	Rationale
Employees.SSN is specified as the primary key of the Employees table.	Tests for SQLitmus's ability to generate primary keys of integer type that respects a configured data generator.
ProjectDetail.Price is configured as a numeric type with a precision of 10 and scale of 2.	Tests for SQLitmus's ability to generate numerical data of a specific precision and scale.
ProjectDetail.EndDate and WorksOn.EndDate are configured to be nullable field.	Tests for SQLitmus's ability to generate null fields of a specified null rate.
ProjectDetail.ProjectId and WorksOn.ProjectId are added to their respective tables.	The javascript library used to generate the test schema, Sequelize, is unable to generate composite foreign keys that do not contain integer fields. This workaround was employed to satisfy that constraint.

Figure 4.2: Table of modifications and rationale

4.2 Test Queries

Based on the test schema, a list of query templates were tested for their performance. The list was selected based on transactional workloads that the test schema was likely to face in a production environment.

The following list of workloads were tested for:

- Hire a new Employee (Insert new Employee record)
- Starting a new Project (Insert new Project record and associated ProjectDetail record)
- Staff an employee on a project (Insert WorksOn record)
- View all active projects at a specified date
- View all employees working on a specific project
- View all projects a specific employee is working on
- View all direct subordinates of a specific employee
- Changing a specified Project record's location (Update a Project record and all associated ProjectDetail and WorksOn records)
- Firing an Employee (Delete Employee record)
- Unstaff an Employee from a project (Delete WorksOn record)
- Ending a project (Delete multiple WorksOn records)

The respective query templates for PostgreSQL and MySQL databases can be found in appendix 1. Despite the differences in the SQL dialects used by both databases, attempts have been made to test both dialects with nearly identical query formats.

4.3 Test Configurations

Figure 24 below specify the row and connection configurations for the test.

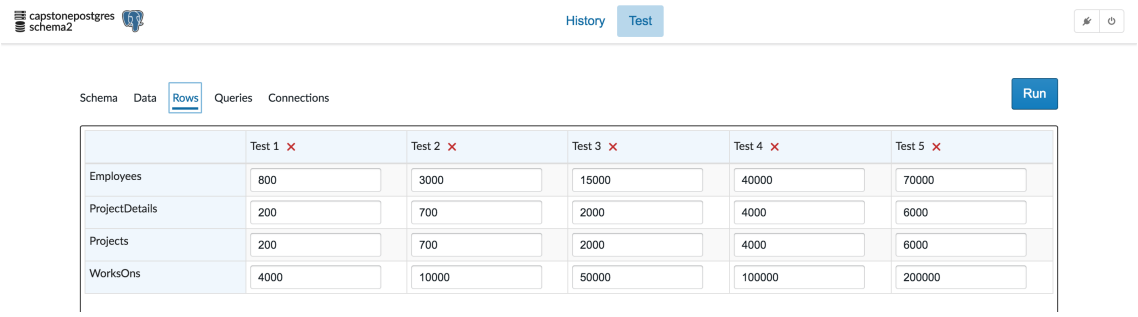


Figure 4.3: Test Schema for pilot study

The rows selected for the test does not reflect the size of typical company records. Rather, they were selected for to test a typical test data generation workload that the target users of SQLitmus are likely to generate. The above specifications generates a total of 518,600 rows and 3,527,200 fields of data.

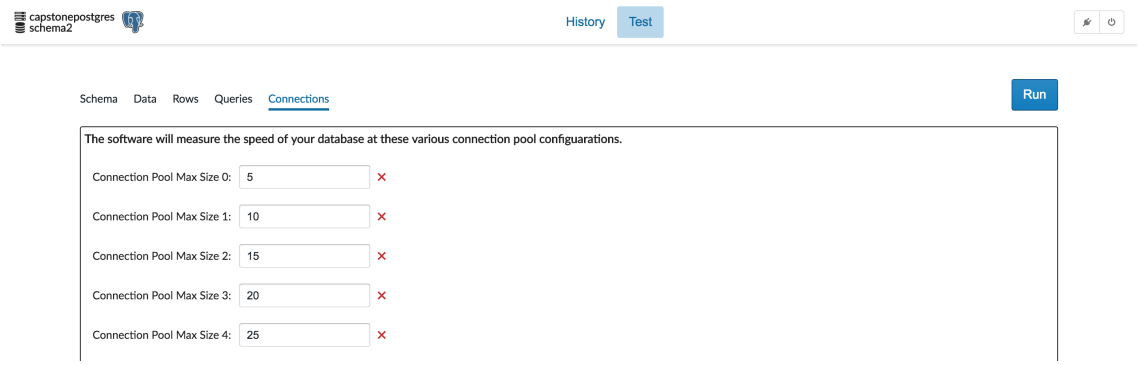


Figure 4.4: Test Schema for pilot study

Similar to the row configurations, the max connection pool configurations displayed in Figure 25 were also selected to represent a typical connection pool workload that an average SQLitmus user is likely to test for.

Chapter 5

Pilot Study

The pilot study ran SQLitmus against three databases: MySQL, PostgreSQL, and MariaDB. All of which were db.t2.micro instances [1] provisioned by Amazon Web Service's Relational Database Service (AWS RDS) .

Each of the three databases were tested over three trials to test for the reliability of SQLitmus's performance analyses.

[1] Each instance contains one virtual CPU, 1GB of RAM, and 20 GB of storage

5.1 Results

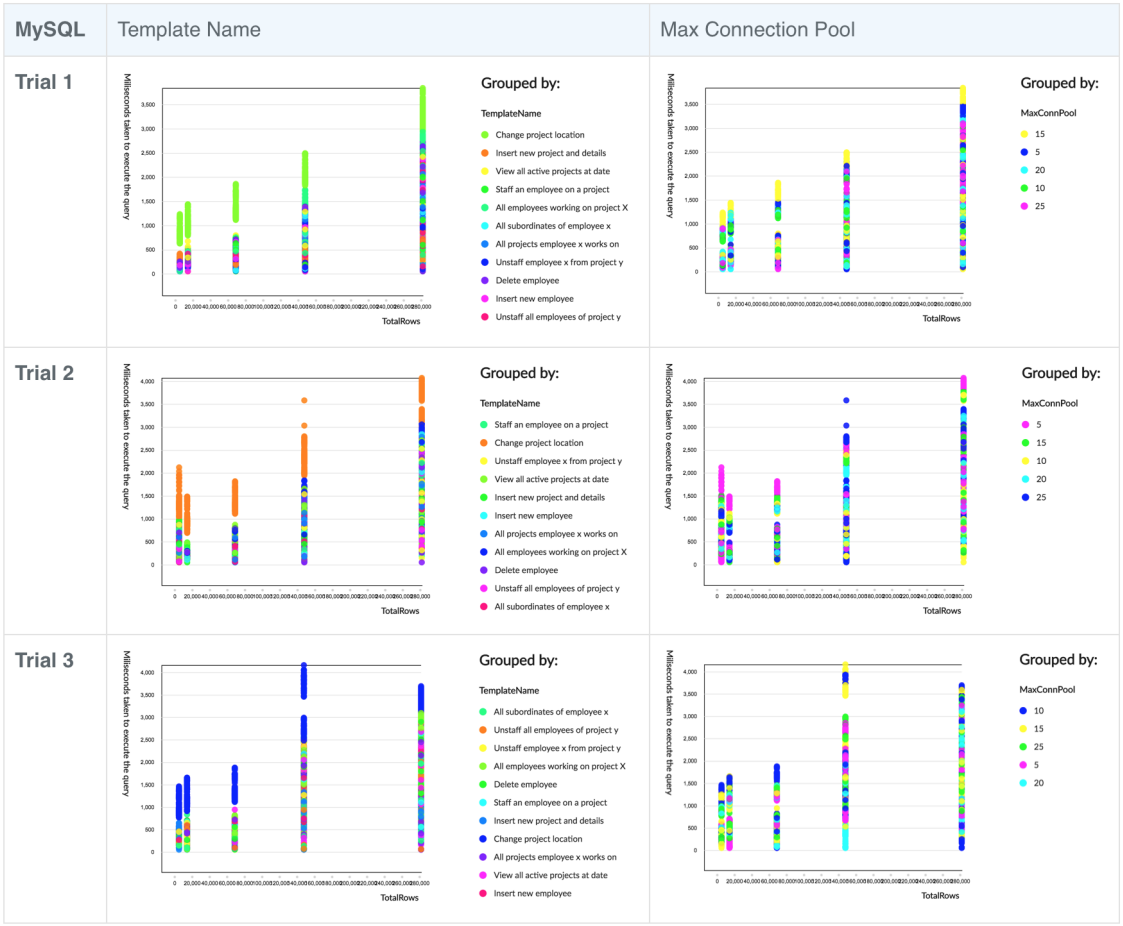


Figure 5.1: Performance results for MySQL

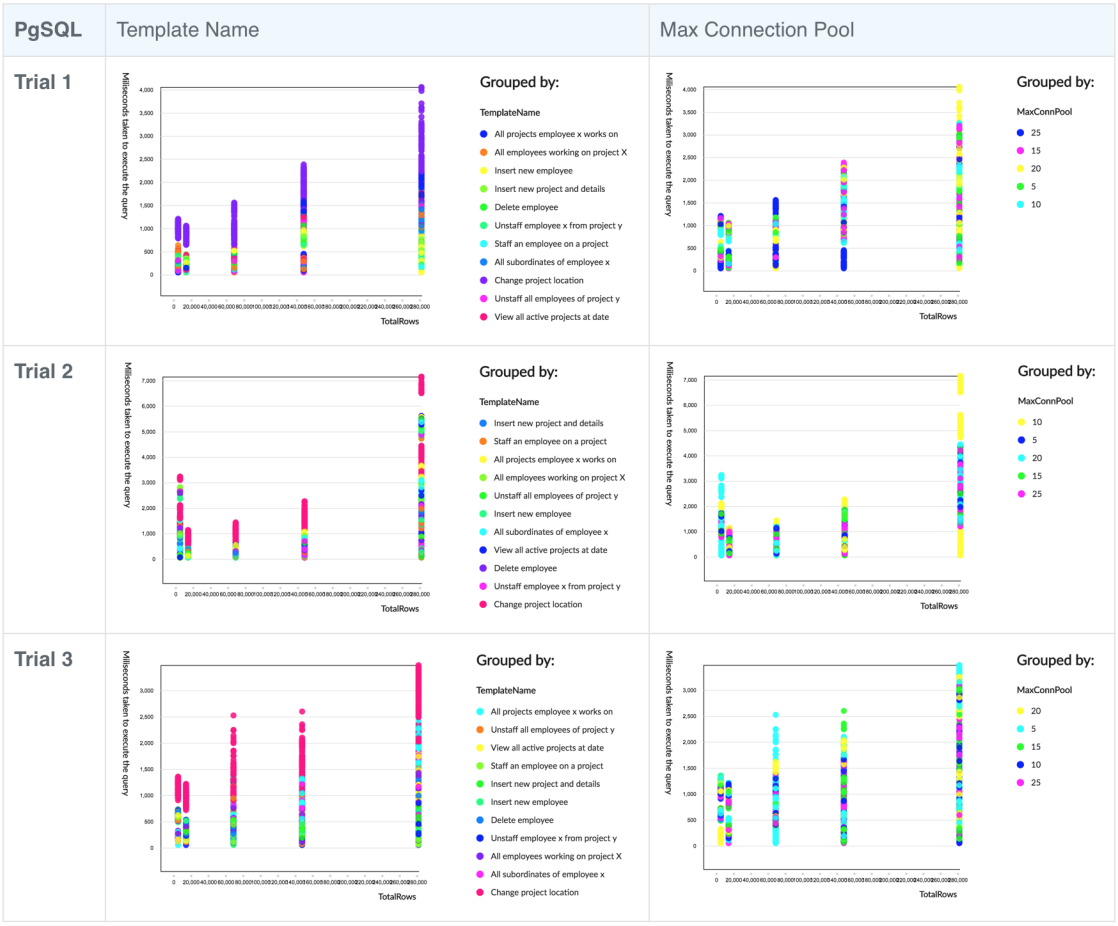


Figure 5.2: Performance results for PostgreSQL

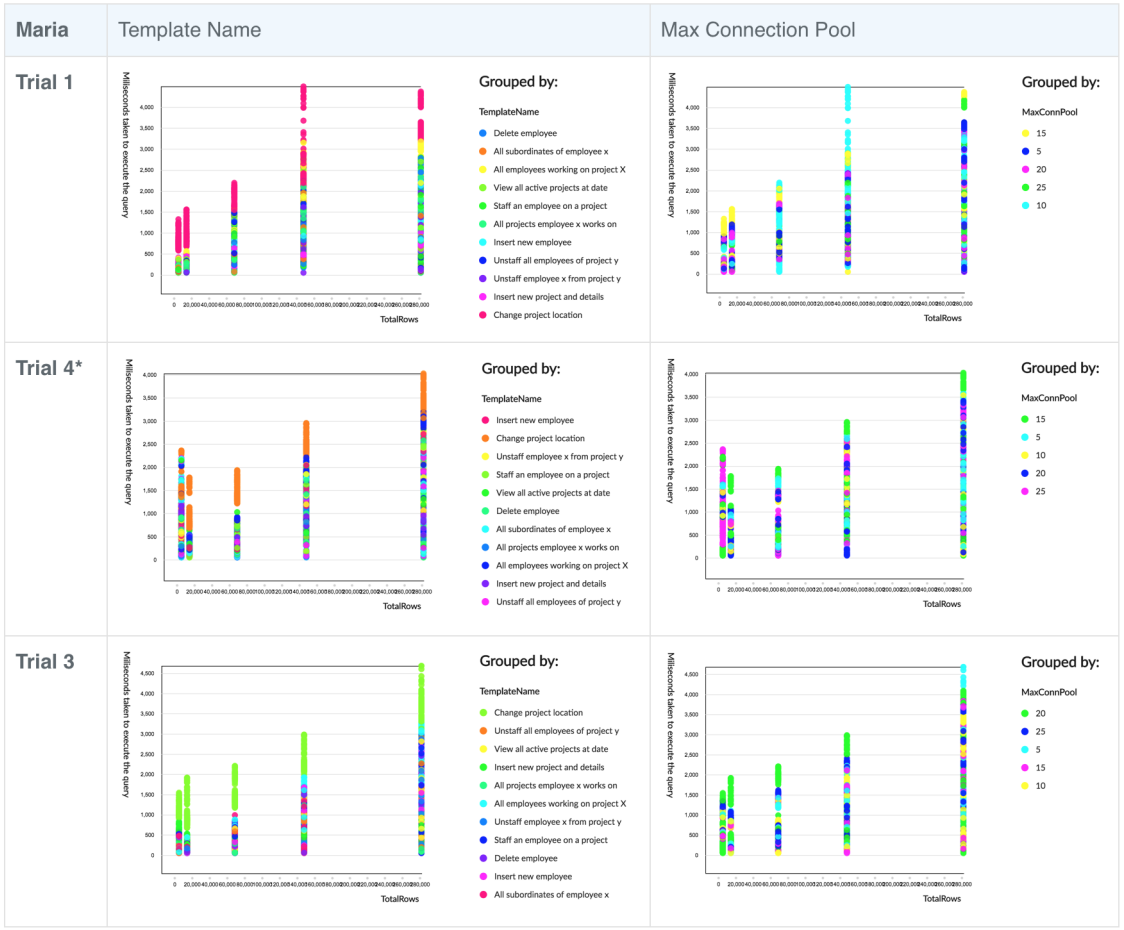


Figure 5.3: Performance results for MariaDB

2

[2]Trial 2 results were discarded as the results were anomalous. The computer went into rest mode during the trial and recorded significantly slower readings. The results are available in Appendix 2.

5.2 Discussion

Let us begin by disambiguating certain terms

- Trial: A single unit of performance analysis conducted by SQLitmus. Each graph in the results section represents an individual trial.
- Test: An environment within a trial that SQLitmus runs queries against. Each column of a graph represents a separate test.

While PostgreSQL's second trial records a maximal response time of 7000 milliseconds, all of the other eight trials ran recorded a maximal response time of 4000 milliseconds. Firstly, these results demonstrates that performance analyses conducted by SQLitmus are reliable, repeatable, and worthy of discussion.

We observe a clear trend of queries taking a longer time to execute as the size of the database increases. This is a consistent finding across all nine trials. While there are some minor exceptions observed which we may attribute to network instability, [3] the observed remains an unmistakable trend.

We are also able to observe the relative performances of each type of query. Across all nine trials, we observe that the time taken to change a project's location is often takes the longest time to execute. While there are also queries that often take a short time to execute, such as inserting a new employee, the graph is far too crowded near the bottom for us to yield any possible intelligence. For cases such as these, developers are able to use the data filtering feature.

[3] The tests were conducted through a Wi-Fi connection.

We also observe no clear trend between the Max Connection Pool Settings and query response time measurements. However, upon visualizing the response times with the Max Connection Pool settings along the X-axis, a faint negative correlation was discovered.

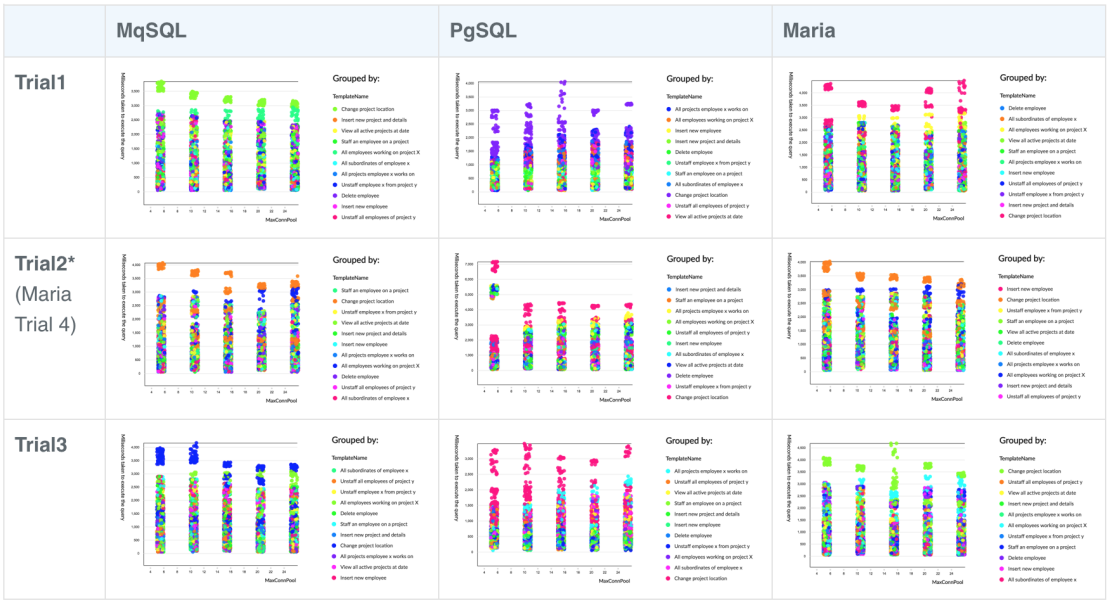


Figure 5.4: Using Max Connection Pool as X-axis

While these results are not conclusive, a contributing cause could be the poor experimental design. The connection pool settings of:

$$5, 10, 15, 20, 25$$

may not have been values distinct enough to affect the database's performance.

Chapter 6

Summary

6.1 Limitations

SQLitmus used many open-sourced libraries to speed up its development. While this is not usually a large concern, SQLitmus is a performance-critical application. To perform more optimally, SQLitmus will have to cut down on the number of libraries used during the data generation process and develop its own components. As it currently stands, SQLitmus is far from being well-tuned and optimized.

Network instability is a confounding variable that SQLitmus has not been able to completely account for. To minimize the impact of network instability, SQLitmus uses a control query to calibrate query response time measurements.

```
1         SELECT 1;
2         --------- Query Response Time Measurement ----
3         SELECT 1;
```

The control query is run before and after measuring any query response time, and the average time taken to run the two control queries is taken as the baseline network instability. SQLitmus then subtracts this baseline value from its respective query response time measurement to minimize the impact of network instability on the recorded results. While it is a simple mechanism, it has drastically improved the quality of performance analyses conducted by SQLitmus.

The results obtained from the pilot test is also limited by the lack of data visualization options provided by the data visualization component.

6.2 Future Work

The following issues are being prioritized for SQLitmus:

1. Shifting in-memory workload to disk storage
2. Faster data generation
3. Query templates that allow zero cardinality drift for singular inserts and deletes
4. Better data visualization

To solve issue 1, SQLitmus is currently exploring the use of flat CSV files for storage. While it is simple to work with csv files, SQLitmus has to tweak its data generation process substantially to ensure that the csv files are created using only a one-time sequential write.

To solve issue 2, SQLitmus is intending to explore the use of algorithms presented

by Alexandrov[1] and Rabl[11]. These algorithms combined with the logarithmic random access times of the PCG RNG will allow SQLitmus to avoid expensive disk reads. This feature becomes substantially more important if the solution to issue 1 were to be successfully implemented.

To solve issue 3, SQLitmus will also capitalize on the PCG RNG to ensure that every single row of data substituted into the template belongs to an actually existing set of data. SQLitmus will use the query generation seed to select a random index of data to regenerate and use the data generation seed to re-generate the same row that it has generated for the database.

To solve issue 4, SQLitmus will incorporate violin plots into the data visualization component. Violin plots are modified box plots with frequecy distributions plotted on each side. Developers thus gain access to summary statistic visualization which includes quartiles and frequency distributions.

6.3 Conclusion

In this paper we presented SQLitmus, a SQL database performance analysis tool. While it is less sophisticated than other advanced data generators, it is far easier to configure, and features a full-blown GUI for managing all aspects of performance analysis - from data generation, and query generation, to environment configuration, and data management. Performance analysis with SQLitmus is demonstrated to be simple to configure thus the tool is likely to suit the demands of developers of small-to-mid sized applications well. SQLitmus also offers a query templating engine that

is more expressive and easier to configure than QGEN[9].

The pilot study of SQLitmus also demonstrated that the tool is capable of generating repeatable and reliable performance analyses of SQL databases. The software recorded clear trends of SQL databases slowing down as their size (amount of data stored) and workload (number of concurrent connections) increased.

Results also revealed performance discrepancies across databases running on identical hardware, data-set, and queries. This shows that SQLitmus can provide developers with intelligence to decide between replaceable databases, queries, and data storage options (e.g., time-stamp vs. date object).

Bibliography

[1] Alexander Alexandrov, Kostas Tzoumas, and Volker Markl. "Myriad: Scalable and Expressive Data Generation". In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1890–1893. ISSN: 2150-8097. DOI: 10.14778/2367502.2367530. URL: http://dx.doi.org/10.14778/2367502.2367530 (cit. on pp. 23, 66).

[2] Nicolas Bruno and Surajit Chaudhuri. "Flexible Database Generators". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 1097–1107. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083719 (cit. on pp. 15, 20).

[3] Jim Gray et al. "Quickly Generating Billion-record Synthetic Databases". In: *SIGMOD Rec.* 23.2 (May 1994), pp. 243–252. ISSN: 0163-5808. DOI: 10.1145/191843.191886. URL: http://doi.acm.org/10.1145/191843.191886 (cit. on p. 14).

[4] Kenneth Houkjær, Kristian Torp, and Rico Wind. "Simple and Realistic Data Generation". In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 1243–

1246. URL: http://dl.acm.org/citation.cfm?id=1182635.1164254 (cit. on pp. 15, 20, 53).

[5] Adam Jacobs. "The Pathologies of Big Data". In: *Queue* 7.6 (July 2009), 10:10–10:19. ISSN: 1542-7730. DOI: 10.1145/1563821.1563874. URL: http://doi.acm.org/10.1145/1563821.1563874 (cit. on p. 24).

[6] Pierre L'Ecuyer. "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure". In: *Math. Comput.* 68.225 (Jan. 1999), pp. 249–260. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-99-00996-5. URL: http://dx.doi.org/10.1090/S0025-5718-99-00996-5 (cit. on p. 23).

[7] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014 (cit. on p. 22).

[8] Meikel Poess and Chris Floyd. "New TPC Benchmarks for Decision Support and Web Commerce". In: *SIGMOD Rec.* 29.4 (Dec. 2000), pp. 64–71. ISSN: 0163-5808. DOI: 10.1145/369275.369291. URL: http://doi.acm.org/10.1145/369275.369291 (cit. on p. 15).

[9] Meikel Poess and John M. Stephens Jr. "Generating Thousand Benchmark Queries in Seconds". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 1045–1053. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316779 (cit. on pp. 13, 17, 21, 67).

[10] Tilmann Rabl et al. "A Data Generator for Cloud-scale Benchmarking". In: *Proceedings of the Second TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems*. TPCTC'10.

Berlin, Heidelberg: Springer-Verlag, 2011, pp. 41–56. ISBN: 978-3-642-18205-1. URL: http://dl.acm.org/citation.cfm?id=1946050.1946054 (cit. on pp. 14, 23).

[11] Tilmann Rabl et al. "Rapid Development of Data Generators Using Meta Generators in PDGF". In: *Proceedings of the Sixth International Workshop on Testing Database Systems*. DBTest '13. New York, NY, USA: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2151-8. DOI: 10.1145/2479440.2479441. URL: http://doi.acm.org/10.1145/2479440.2479441 (cit. on p. 66).

[12] John M. Stephens and Meikel Poess. "MUDD: A Multi-dimensional Data Generator". In: *SIGSOFT Softw. Eng. Notes* 29.1 (Jan. 2004), pp. 104–109. ISSN: 0163-5948. DOI: 10.1145/974043.974060. URL: http://doi.acm.org/10.1145/974043.974060 (cit. on pp. 15, 17).

Appendix A

Query Templates

```
# Insert new employee (MySQL)
DELETE FROM Employees WHERE SSN = ${Employees.SSN};
${BEGIN.DELIMITER}
INSERT INTO Employees VALUES
(null,FROM_UNIXTIME(CEIL(${Employees.EmploymentDate}/1000)),
${Employees.FirstName},${Employees.LastName},
${Employees.SSN}, ${Employees.RANDROW});
${END.DELIMITER}
DELETE FROM Employees WHERE SSN = ${Employees.SSN};

# Insert new employee (PostgreSQL)
DELETE FROM "Employees" WHERE "SSN" = ${Employees.SSN};
${BEGIN.DELIMITER}
INSERT INTO "Employees"  VALUES
(DEFAULT, to_timestamp(CEIL(${Employees.EmploymentDate}/1000)),
${Employees.FirstName},${Employees.LastName}, ${Employees.SSN},
${Employees.RANDROW});
${END.DELIMITER}
DELETE FROM "Employees" WHERE "SSN" = ${Employees.SSN};

# Insert new project and details (MySQL)
DELETE FROM Projects WHERE
Name = ${Projects.Name} AND Location = ${Projects.Location};
DELETE FROM ProjectDetails WHERE
ProjectName = ${Projects.Name} AND ProjectLocation = ${Projects.Location};
```

```
${BEGIN.DELIMITER}
INSERT INTO Projects VALUES
(null,${Projects.Name},${Projects.Location}, ${Projects.Priority});
INSERT INTO ProjectDetails Values (null,
FROM_UNIXTIME(CEIL(${ProjectDetails.StartDate}/1000)),
FROM_UNIXTIME(CEIL(${ProjectDetails.EndDate}/1000)),
${ProjectDetails.Price}, ${ProjectDetails.ManHours},
${Projects.Name}, ${Projects.Location},
(SELECT max(id) id FROM Projects));


# Insert new project and details (PostgreSQL)
DELETE FROM "Projects" WHERE
"Name" = ${Projects.Name} AND "Location" = ${Projects.Location};
DELETE FROM "ProjectDetails" WHERE
"ProjectName" = ${Projects.Name} AND "ProjectLocation" = ${Projects.Location};
${BEGIN.DELIMITER}
INSERT INTO "Projects" VALUES
(default,${Projects.Name},${Projects.Location}, ${Projects.Priority});
INSERT INTO "ProjectDetails" Values (DEFAULT,
to_timestamp(CEIL(${ProjectDetails.StartDate}/1000)),
to_timestamp(CEIL(${ProjectDetails.EndDate}/1000)),
${ProjectDetails.Price}, ${ProjectDetails.ManHours},
${Projects.Name},${Projects.Location},
(SELECT "id" FROM "Projects" ORDER BY id DESC LIMIT 1));

#Staff an employee on a project (MySQL)
DELETE FROM WorksOns WHERE
ProjectName = ${Projects.Name} AND ProjectLocation = ${Projects.Location};
${BEGIN.DELIMITER}
INSERT INTO WorksOns VALUES (null,
FROM_UNIXTIME(CEIL(${WorksOns.StartDate}/1000)),
FROM_UNIXTIME(CEIL(${WorksOns.EndDate}/1000)),
${Projects.Name}, ${Projects.Location},
${Employees.RANDROW},${Projects.RANDROW});

#Staff an employee on a project (PostgreSQL)
INSERT INTO "WorksOns" VALUES (DEFAULT,
to_timestamp(CEIL(${WorksOns.StartDate}/1000)),
to_timestamp(CEIL(${WorksOns.EndDate}/1000)),
${Projects.Name}, ${Projects.Location},
CEIL(random()*${Employees.numRows}),
CEIL(random()*${Projects.numRows}));

```

```
#View all active projects at date (MySQL)
SELECT Name, Location, StartDate, EndDate, Price, ManHours
FROM ProjectDetails JOIN Projects
ON Projects.Name=ProjectDetails.ProjectName
AND Projects.Location=ProjectDetails.ProjectLocation
WHERE StartDate < FROM_UNIXTIME(CEIL(${ProjectDetails.StartDate}/1000))
AND EndDate > FROM_UNIXTIME(CEIL(${ProjectDetails.StartDate}/1000));

#View all active projects at date (PostgreSQL)
SELECT "Name", "Location", "StartDate", "EndDate", "Price", "ManHours"
FROM "ProjectDetails" , "Projects"
WHERE "ProjectName" = "Name"
AND "ProjectLocation" = "Location"
AND "StartDate" < to_timestamp(CEIL(${ProjectDetails.StartDate}/1000))
AND "EndDate" > to_timestamp(CEIL(${ProjectDetails.StartDate}/1000));

#All employees working on project X (MySQL)
SELECT Employees.id, FirstName, LastName,
Employees.SSN, ProjectName, ProjectLocation
FROM Employees, WorksOns, Projects
WHERE WorksOns.EmployeeId = Employees.id
AND WorksOns.ProjectName LIKE Projects.Name
AND WorksOns.ProjectLocation LIKE Projects.Location
AND Projects.id = ${Projects.RANDROW};

#All employees working on project X (PostgreSQL)
SELECT "Employees".id, "FirstName", "LastName",
"SSN", "ProjectName", "ProjectLocation"
FROM "Employees" , "WorksOns", "Projects"
WHERE "Employees".id = "EmployeeId"
AND "WorksOns"."ProjectName" = "Projects"."Name"
AND "WorksOns"."ProjectLocation" = "Projects"."Location"
AND "Projects".id = ${Projects.RANDROW};

#All projects employee x works on (MySQL)
SELECT Projects.id, Name, Location, EmployeeId
FROM Projects JOIN WorksOns
ON Name = ProjectName
AND Location = ProjectLocation
WHERE WorksOns.EmployeeId=${Employees.RANDROW};

#All projects employee x works on (PostgreSQL)
SELECT "Projects".id, "Name", "Location", "EmployeeId"
FROM "Projects" , "WorksOns"
WHERE "Name" = "ProjectName"
AND "Location" = "ProjectLocation"
```

```
116         AND "WorksOns"."EmployeeId" = ${Employees.RANDROW};
117
118         # All subordinates of employee x (MySQL)
119         SELECT * FROM Employees WHERE WorksFor=${Employees.RANDROW};
120
121         # All subordinates of employee x (PostgreSQL)
122         SELECT * FROM "Employees" WHERE "WorksFor" = ${Employees.RANDROW};
123
124         # Change project location (MySQL)
125         ${BEGIN.DELIMITER}
126         UPDATE Projects SET Location = 'NEW RANDOM LOCATION'
127         WHERE Projects.Name=${Projects.Name}
128         AND Projects.Location=${Projects.Location};
129         UPDATE WorksOns SET ProjectLocation = 'NEW RANDOM LOCATION'
130         WHERE WorksOns.ProjectName=${Projects.Name}
131         AND WorksOns.ProjectLocation=${Projects.Location};
132         UPDATE ProjectDetails SET ProjectLocation = 'NEW RANDOM LOCATION'
133         WHERE ProjectDetails.ProjectName=${Projects.Name}
134         AND ProjectDetails.ProjectLocation=${Projects.Location};
135         ${END.DELIMITER}
136         UPDATE Projects SET Location = ${Projects.Location}
137         WHERE Projects.Name=${Projects.Name}
138         AND Projects.Location='NEW RANDOM LOCATION';
139         UPDATE WorksOns SET ProjectLocation = ${Projects.Location}
140         WHERE WorksOns.ProjectName=${Projects.Name}
141         AND WorksOns.ProjectLocation='NEW RANDOM LOCATION';
142         UPDATE ProjectDetails SET ProjectLocation = ${Projects.Location}
143         WHERE ProjectDetails.ProjectName=${Projects.Name}
144         AND ProjectDetails.ProjectLocation='NEW RANDOM LOCATION';
145
146         # Change project location (PostgreSQL)
147         ${BEGIN.DELIMITER}
148         UPDATE "Projects" SET "Location" = 'NEW RANDOM LOCATION'
149         WHERE "Projects"."Name"=${Projects.Name}
150         AND "Projects"."Location"=${Projects.Location};
151         UPDATE "WorksOns" SET "ProjectLocation" = 'NEW RANDOM LOCATION'
152         WHERE "WorksOns"."ProjectName"=${Projects.Name}
153         AND "WorksOns"."ProjectLocation"=${Projects.Location};
154         UPDATE "ProjectDetails" SET "ProjectLocation" = 'NEW RANDOM LOCATION'
155         WHERE "ProjectDetails"."ProjectName"=${Projects.Name}
156         AND "ProjectDetails"."ProjectLocation"=${Projects.Location};
157         ${END.DELIMITER}
158         UPDATE "Projects" SET "Location" = ${Projects.Location}
159         WHERE "Projects"."Name"=${Projects.Name}
```

```
160         AND "Projects"."Location"='NEW RANDOM LOCATION';
161         UPDATE "WorksOns" SET "ProjectLocation" = ${Projects.Location}
162         WHERE "WorksOns"."ProjectName"=${Projects.Name}
163         AND "WorksOns"."ProjectLocation"='NEW RANDOM LOCATION';
164         UPDATE "ProjectDetails" SET "ProjectLocation" = ${Projects.Location}
165         WHERE "ProjectDetails"."ProjectName"=${Projects.Name}
166         AND "ProjectDetails"."ProjectLocation"='NEW RANDOM LOCATION';
167
168         # Delete employee (MySQL)
169         INSERT INTO Employees Values (null,
170         FROM_UNIXTIME(CEIL(${Employees.EmploymentDate}/1000)),
171         ${Employees.FirstName},${Employees.LastName},
172         ${Employees.SSN}, ${Employees.RANDROW});
173         ${BEGIN.DELIMITER}
174         DELETE FROM Employees WHERE SSN = ${Employees.SSN};
175
176         # Delete employee (PostgreSQL)
177         INSERT INTO "Employees" Values (DEFAULT,
178         to_timestamp(CEIL(${Employees.EmploymentDate}/1000)),
179         ${Employees.FirstName},${Employees.LastName},
180         ${Employees.SSN}, ${Employees.RANDROW});
181         ${BEGIN.DELIMITER}
182         DELETE FROM "Employees" WHERE "SSN" = ${Employees.SSN};
183
184         # Unstaff employee x from project y (MySQL)
185         INSERT INTO WorksOns Values (null,
186         FROM_UNIXTIME(CEIL(${WorksOns.StartDate}/1000)),
187         FROM_UNIXTIME(CEIL(${WorksOns.EndDate}/1000)),
188         ${Projects.Name},${Projects.Location},
189         ${Employees.RANDROW},${Projects.RANDROW});
190         ${BEGIN.DELIMITER}
191         DELETE FROM WorksOns WHERE ProjectName=${Projects.Name}
192         AND ProjectLocation=${Projects.Location}
193         AND EmployeeId=${Employees.RANDROW};
194         ${END.DELIMITER}
195
196         # Unstaff employee x from project y (PostgreSQL)
197         INSERT INTO "WorksOns" Values (DEFAULT,
198         to_timestamp(${WorksOns.StartDate}),
199         to_timestamp(${WorksOns.EndDate}),
200         ${Projects.Name}, ${Projects.Location},
201         ${Employees.RANDROW}, ${Projects.RANDROW});
202         ${BEGIN.DELIMITER}
```

```
203         DELETE FROM "WorksOns" WHERE "ProjectName"=${Projects.Name}
204         AND "ProjectLocation"=${Projects.Location}
205         AND "EmployeeId"=${Employees.RANDROW};
206         ${END.DELIMITER}
207
208
209         # Unstaff all employees of project y (MySQL)
210         DELETE FROM WorksOns
211         WHERE ProjectName=${Projects.Name}
212         AND ProjectLocation=${Projects.Location};
213
214
215         # Unstaff all employees of project y (PostgreSQL)
216         DELETE FROM "WorksOns"
217         WHERE "ProjectName"=${Projects.Name}
218         AND "ProjectLocation"=${Projects.Location};
```

Appendix B

Excluded MariaDB Trial 2 results

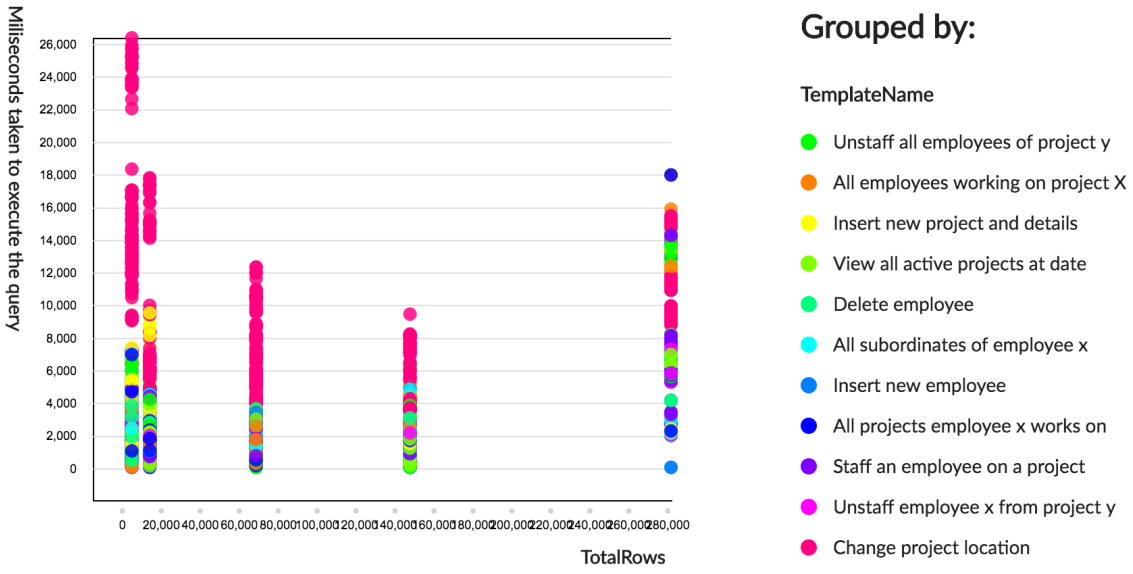


Figure B.1: Using Total Rows as X-axis, group by Template Name

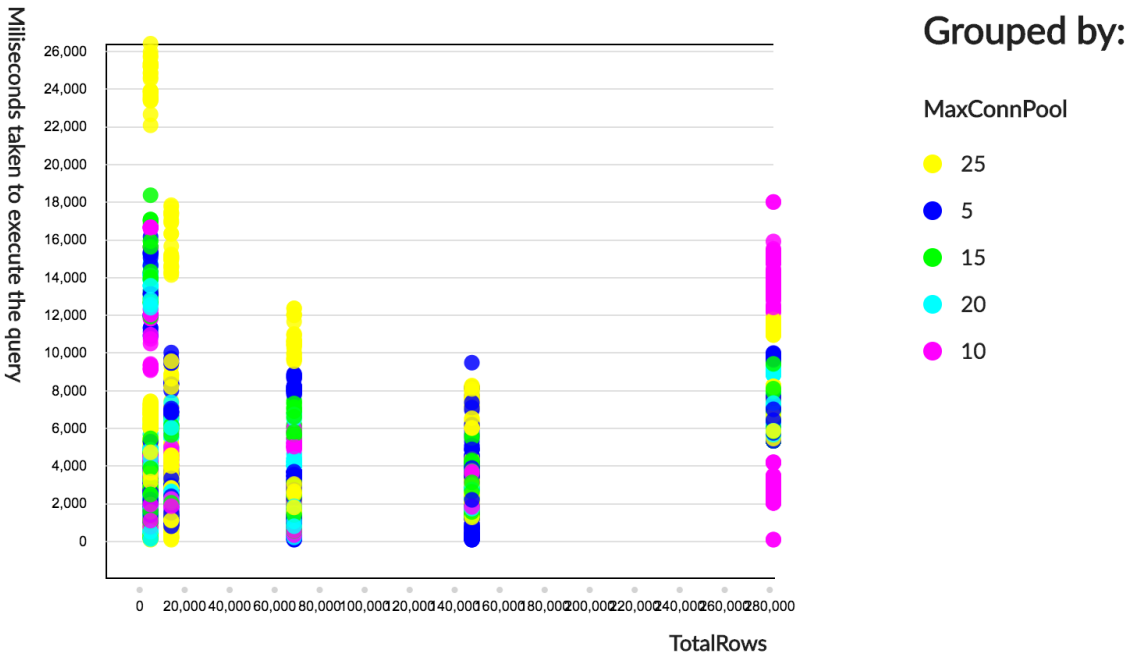


Figure B.2: Using Total Rows as X-axis, group by Max Connection Pool

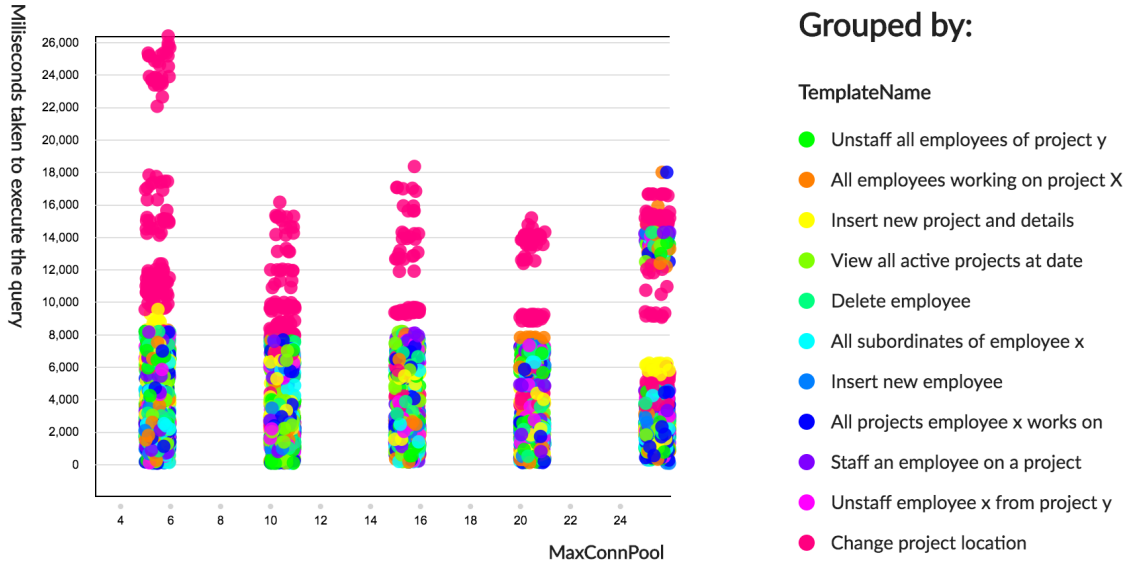


Figure B.3: Using Max Connection Pool as X-axis, group by Template Name