Москва 2020

Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский университет «Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
«Прикладная математика и информатика»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект на тему

РАЗРАБОТКА ИНТЕРФЕЙСАСРАВНЕНИЯ И АНАЛИЗА ПРОИЗВОДИТЕЛЬНОСТИ РЕВИЗИЙ CLICKHOUSE

Выполнила студентка группы БПМИ164, 4 курса,

Обрезкова Дарья Валерьевна

Руководитель ВКР:

Руководитель группы разработки ClickHouse,

Миловидов Алексей Николаевич

Оглавление

Аннотация

Аннотация - В настоящее время мало внимания уделяется проблеме визуализации тестовых данных в сети, а также разработке программ для этого. Целью этого проекта является создание веб-интерфейса для быстрого понимания результатов системы тестирования и отслеживания качества работы над кодом системы управления базами данных (СУБД) ClickHouse. Используя алгоритмы классификации, языка программирования JavaScript, а также новые платформы и библиотеки для визуализации данных, была создана система, которая показывает результаты тестирования ревизий ClickHouse. Данная система визуализации в виде набора графиков, диаграмм и таблиц, а также самого интерфейса для удобного просмотра отображаемой информации может быть использована как разработчиками, так и другими специалистами из сферы IT, так как запуск и использование системы не требует специфичных навыков программирования. Пользователь может просмотреть результаты тестирования и выяснить, сколько тестов было пройдено, завершилось с ошибкой или пропущено; насколько улучшен код в текущей версии программы по сравнению с предыдущей версией.

Помимо этого, интерфейс спроектирован так, чтобы в дальнейшим принимать запросы с backend-а и визуализировать результаты тестирования более 2 версий кода. В результате работы было создано статичное веб-приложение с помощью стека технологий: язык программирования JavsScript, программная платформа Node.js, языки стиля и разметки HTML5 и CSS3, фреймворков Vue.js и Nuxt.js.

Данное решение не только решает вопрос визуализации тестов в open-sourse проекте ClickHouse, но и позволяет по-новому взглянуть на отображение результатов тестирования при разработке IT-продукта.

Работа содержит 41 страницу, 3 главы, 13 рисунков, 20 источников

Ключевые слова — *JavaScript, Vue, графики, визуализация, веб-разработка, статическое приложение, система управления базами данных, СУБД, ClickHouse*

Abstract

Abstract - Currently, very little attention is paid to the problem of visualizing test data on the network, as well as developing programs for this. The goal of this project is to create a web interface for quick understanding of the results of the testing system and tracking the quality of work on the code of the database management system (DBMS) ClickHouse. Using classification algorithms, JavaScript programming language, as well as new platforms and libraries for data visualization, a system was created that shows the results of testing ClickHouse revisions. This visualization system in the form of a set of graphs, charts and tables can be used both by developers and other IT specialists, since starting and using the system does not require specific programming skills. The user can view the test results and find out how many tests have been completed, failed or skipped; how much the code was improved in the current version of the program compared to the previous version. In addition, the interface is designed to accept requests from the backend in the future and visualize the test results of more than 2 versions of the code. As a result of work, a static web application was created using a technology stack: JavsScript as programming language, software platform Node.js, markup and style languages HTML5 and CSS3, Vue.js and Nuxt.js frameworks. This solution not only solves the question of visualization of tests in the open-source project ClickHouse, but also

allows you take a fresh look at the display of test results in the development of an IT product.

The work contains 41 pages, 3 chapters, 13 drawings, 20 sources.

Keywords — *JavaScript, Vue, graphs, visualization, web development, static application, database management system, DBMS, ClickHouse*

Основные понятия

1. База данных — это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе. База данных обычно управляется системой управления базами данных (СУБД). [1]
2. ClickHouse — это созданный компанией Яндекс open-sourse проект, представляющий собой столбчатую систему управления базами данных (СУБД) для онлайн обработки аналитических запросов (OLAP). [2]
3. Статичное приложение (Static Generated) – веб-приложение, которое может быть размещено для работы на статичном ресурсе (Netlify, GitHub Pages и проч.) [3]
4. Открытое программное обеспечение (англ. open-source software) программное обеспечение с открытым исходным кодом
5. Веб-компоненты — переиспользуемые виджеты пользовательского интерфейса, которые создаются с помощью открытых веб-технологии, которые

позволяют создавать новые, пользовательские HTML-элементы со своими свойствами, методами, инкапсулированными DOM и стилями. [4],[5]

6. Бандл (bundle) - это совокупность каких-либо программных данных (файлов), объеденных по какому-либо признаку.

7. Ревизия кода - слияние воедино разных версий кода, их тестирование и проведение анализа результативности новой полученной версии.

8. SQL запрос - запрашивание у базы данных информации на специальном языке запросов (в нашем случаи, на языке запросов к Clickhouse) с определенными параметрами.

Введение

В настоящее время в интернете можно видеть немало визуализированных массивов данных. При размещении статистических данные, зачатую их подкрепляют графиками и диаграммами, но в основном это статичные картинки, а не зависимые от данных изменяемые веб-компоненты. Совсем стороной обходят frontend часть тестирующих систем несмотря на то, что подобный функционал используется при разработке достаточно часто. Многие разработчики используют функциональное и нефункциональное тестирование в своей работе, что позволяет избежать ошибок, предотвратить сбои и увеличить безопасность конечного продукта. Однако, многие существующие тестирующие системы не стремятся делать упор на анализ и визуализацию данных. Чаще всего их работа ограничивается отображением прохождение конкретного теста и общего числа тестов. Как правило, подобные интерфейсы

не очень удобны для использования. Если результаты тестирования представлены в виде большого количества данных, которые не были верно предобработаны, их визуализация неоднозначна и плохо структурирована, то в них сложно искать необходимую информацию. В данной работе главной задачей становится обработка результатов тестовых данных и их последующая информативная визуализация. Помимо этого, будет рассмотрена специфическая сборка frontend части проекта, отвечающая за интерфейс, которая позволит создать статичное веб-приложение.

Цели и задачи

Цель проекта – это создание системы для анализа и сравнения ревизий ClickHouse, быстрого понимания результатов тестирующей системы и отслеживания качества работы над кодом. Работа с данными здесь — это обработка файлов с результатами тестирования двух версий кода, представление информации из них в виде json файла и его отправка на frontend часть. Со стороны интерфейса работа с данными представляет собой получение json файла, его обработка и представление данных в нужном формате для визуализации.

Интерфейс должен быть одновременно минималистичным и информативным, чтобы пользователь смог быстро оценить прогресс работы над версиями кода, а также получить подробную информацию о результатах при необходимости.

В ходе работы были поставлены 3 главные задачи.

1. Провести обработку данных о результатах тестирования ревизий ClickHouse.

2. Разработать набора визуальных элементов (графиков, таблиц и т.п.) для удобного и информативного отображения полученных данных

3. Разработать веб-интерфейс сравнения и анализа производительности ревизий ClickHouse.

В том числе необходимо было провести исследование существующих решений и технологий, в также изучить вопросы визуализации данных (в том числе результатов тестирования программного кода) в сети Интернет; классификации данных (в частности результатов тестирования программного кода). При этом необходимо было изучить материалы, необходимые для разработки визуальной части веб-интерфейса и сборки статического веб-приложения

Согласно поставленным задачам, на странице интерфейса должны отображаться:

- пройденные, не пройденные и пропущенные тесты, а также их количество;
- графики соотношения пройденных, не пройденные и пропущенных тестов;
- сравнения времени прохождения тестов текущей версии кода с предыдущими;
- таблицы с результатами тестирования 2 версий кода, причем отличающиеся тесты должны быть представленные в разных таблицах/графиках и т.д.;
- дополнительные графики, отражающие результаты тестирования.

В данной работе будет подробно описано решение проблемы визуализации результатов тестирования с помощью веб-разработки в целом, и проблемы реализации программы для ревизий ClickHouse.

Обзор существующих решений

В ходе работы были проанализированы имеющиеся решения и выяснено, какие схожие продукты существуют и какие у них особенности.

Самыми популярными инструментами для визуализации результатов тестирования на данный момент являются Azure DevOps [6], TeamCity [7], Jenkins [8] и внутренние интерфейсы систем контроля версий, были рассмотрены наиболее известные – GitHub и GitLab. Так как ClickHouse – это open-sourse проект, в данной работе не будут рассматриваться внутренние инструменты Яндекса для визуализации.

Azure DevOps и TeamCity – платные для коммерческого использования продукты с понятной и красивой визуализацией и системой отслеживания версий. В этих продуктах используются графики, программы имеют удобный дизайн, однако для проекта рассматривались только бесплатные продукты.

Jenkins – это open-sourse проект с несовременным дизайном, однако его использование бесплатно. Графики не очень удобные, но информативные – этот важный аспект был учтен при написания собственной программы. Так как решается проблема визуализации именно тестов ClickHouse, излишний функционал Jenkins будет лишь мешать при использовании. При этом, Azure DevOps, TeamCity, Jenkins не позволят нам быстро и эффективно сравнивать ClickHouse с другими продуктами.

Решение имеющейся проблемы визуализации предлагается в GitHub и GitLab, однако в них не очень хорошо проработаны и детализированы

результаты тестирования кода. В интерфейсе системы отображается, сколько тестов было запущено и какие из них пройдены, а какие нет.

В результате анализа и сравнения существующих инструментов, было решено создать собственную программу для визуализации результатов тестирования, которая бы специально подходила для работы в ClickHouse и не содержала бы излишнего для работы функционала.

Глава 1 - Выбор методов решения

Процесс работы над данным проектом можно разделить на 2 смысловые части: работа с данными – результатами тестирования и создание интерфейса. Для лучшего понимая, два данных процесса будут описаны в данной главе отдельно.

1.1 Работа с данными

На этапе анализа и классификации данных [9] предполагается работа с 3 типами тестов: 1) функциональные тесты SQL запросов 2) тесты производительности SQL запросов 3) интеграционные тесты.

Функциональные тесты - набор запросов к базе данных или sh скрипт, для которого проверяется результат тестирования.

Тесты производительности - набор запросов к базе данных, для которого проверяется производительность.

Интеграционные тесты - тесты, разворачивающие произвольное окружение, в котором кластеры ClickHouse работают совместно с другими системами, и проверяют реализованные в них сценарии.

Классификация SQL-запросов имеет некоторые проблемы [10], поэтому для работы с тестированием SQL-запросов была использована следующая группировка:

1. По типу запросов - SELECT, INSERT, DELETE запросы;

2. По статусу тестов - пройден / не пройден / пропущен;

3. По времени прохождения тестов с установленным порогом p, начиная с которого тест считается «ускорившимся» или «замедлившимся»;

4. По количеству запусков запросов во время тестирования;

5. По разнице во времени прохождения теста на клиенте и на сервере.

Все исходные данные хранятся в разных файлах в форматах .txt, .rep и .tsv (tab separated values). Тесты ClickHouse определяют правильность выводимых результатов SQL запросов и скорость их выполнения.

Для лучшего понимания данных, было решено организовать результаты тестирования в структуру (набор файлов), которая позднее будет записана в json файл, и классифицировать их.

Для работы с данными были написаны программы на языке Python3. Выбор обусловлен использова данного языка в текущих программах для обработки файлов в системе ClickHouse. Помимо этого, использование Python3 позволяет быстро написать кода для обработки файловых данных.

Работа с данными в интерфейсе производилась исключительно с помощью языка программирования JavaScript и имеющихся в нем встроенных функций.

1.2 Работа с интерфейсом

Для реализации интерфейса в качестве главного языка программирования был выбран интерпретируемый язык JavaScript, так как его функционал лучше других языков программирования подходит для создания веб-приложений.

JavaScript встроен в каждый браузер, поэтому при открытии в нем работающей программы не возникнет проблемы интерпретации написанного кода [11]. После сборки проекта код конвертируется в версию, которая поддерживается наиболее популярными современными браузерами (Chrome, Firefox и др.).

В качестве фреймворков для работы были выбраны Vue.js[12] и Nuxt.js[13].

Vue.js — это быстрорастущая платформа с открытым исходным кодом с лицензией MIT. Его основным преимуществом является удобный синтаксис, быстрый рендеринг страницы и простая и быстрая интеграция с библиотекой для визуализации диаграмм и графиков, которая была использована. Данный фреймворк не может сравниться по количеству использований с самым популярным на данный момент фреймворком React.js, разработанным и поддерживаемым Facebook. Однако Vue.js имеет много преимуществ перед ним - он намного удобнее в использовании, а написанные на нем проекты не перегружены кодом. Данный фреймворк используется компаниями GitLab, Alibaba, Apple, Google и другие.

Так как отрисовка графиков на чистом JavaScript + HTML5, CSS3 была бы слишком затратная по времени, было решено использовать одну из библиотек для построения графиков.

Были рассмотрены самые популярные библиотеки для рисования графиков и charts — это D3.js [14], [15], Chart.js [16] и Highcharts.js [14], [17]. Все три библиотеки бесплатны для некоммерческого использования, но Highcharts содержит самое разнообразное количество графиков, проста в использовании и легко интегрируется в код на Vue.js. Кроме того, данную библиотеку официально использует Яндекс – ментейнер ClickHouse.

Для работы с хранилищем данных store, о котором будет подробнее сказано в Главе 2 «Работа с данными и создание проекта», нам понадобится

библиотека Vuex, специально созданная для написания хранилища на Vue.js. Данная библиотека помогает в создании части приложения, которая отвечает за хранение основных данных, которые используются в проекте. Vuex – это хранилищем данных, к которому могут обращаться все компоненты приложения, которое защищает состояние store от изменения непредсказуемым образом [18].

В силу технологий, которые используются ClickHouse, было необходимо, чтобы веб-интерфейс представлял из себя статичную страницу, зависящую от статичных файлов (картинок, данных и т.д.), которая может быть загружена на GitHub и обновлялась бы в случае изменений окружения. Главной задачей при разработке было создание веб-приложения, для которого работы которого не требуется сервер. В качестве генерации статичной страницы был использован фреймворк Nuxt.js. Он не только собирает бандл для размещения на статическом ресурсе, но и упрощает разработку и оптимизирует готовое приложение. Данный фреймворк специально создан для работы с приложениями, написанными на Vue.js. Он требует немного другого подхода в разработке при написании кода, создания иерархии папок и файлов проекта, сборки и написания config файлов – файлов конфигурации в отличии от простого проекта, написанного на Vue.js без использования Nuxt.js. Взамен мы получаем готовое для использования статическое веб-приложение, которое представляет из себя набор из статичного HTML файла (страница, которая будет отображаться в браузере), набора статичных файлов (данных, картинок и т.п.), и кода JavaScript.

1.3 Выводы и результаты

В ходе работы был проведен полноценный анализ существующих решений и инструментов для разработки веб-приложений. На основании проведенного

анализа был обоснован выбор технологий для реализации данного проекта. В данный стек технологий вошел язык программирования JavaScript как общепризнанный главный язык при разработке frontend-части веб-приложения, языки разметки и стилей HTML5 и CSS3. Так же при разработке использовались фреймворки Vue.js (в силу его быстрой работы и лаконичной организации структуры кода) и Nuxt.js (как инструмент сборки статичных веб-приложений, разработанный специально для кода на Vue.js). Также были использованы библиотеки Vuex и Highcharts.js для работы с хранилищем и постройки графиков и диаграмм соответственно.

Глава 2 – Работа с данными и создание проекта

Поскольку разрабатываемое приложение должно было быть статичным и работать без сервера, как таковой backend-части, к которой посылаются запросы для обмена данными, также не предполагалось.

Однако, в проекте присутствует предобработка данных для передачи на frontend-часть, анализ которой более подробно изложен в данном разделе.

2.1 Работа с файлами

Структура данных с результатами тестирования состоит из набора файлов и папок, в которых содержатся файлы в форматах .txt, .rep, .tsv, .svg и другие. Однако, нас интересуют только следующие 13 файлов.

all-queries.tsv

Содержит информацию о всех запросах, которые посылаются в базу данных ClickHouse при тестировании. Запросы уникальны, однако несколько

запросов могут быть объединены в 1 тест. Такие запросы хранятся отдельно друг от друга, но имеют одинаковое поле testName. Файл не содержит запросы, которые были пропущены или чье тестирование закончилось с ошибкой.

bad-tests.tsv

Содержит информацию о количестве нестабильных (unstable), значительно изменившихся относительно времени тестирования (changes in performance) запросов и их общем количестве для каждого из тестов, содержащих такие запросы.

changed-perf.tsv

Содержит информацию о запросах, производительность которых была отмечена как существенная. Для интерпретации результатов, во время тестирования строится рандомизированное распределение для разницы медианного времени между старым и новым временем при нулевой гипотезе о том, что распределение производительности одинаково. Наблюдаемая разница в производительности считается существенной, если она выше 5% и выше 95-го перцентиля распределения рандомизации. [19]

compare.log

Содержит информацию (логи) об изменениях в текущей версии кода – последнего коммита.

left-commit.txt

Содержит информацию о предыдущей версии кода: номер коммита, кто написал данный код, дата его публикации и дополнительная информация.

right-commit.txt

Содержит ту же информацию, что и `left-commit.txt`, но о текущем коммите.

run-errors.tsv

Содержит информацию об ошибках, которые произошли во время тестирования.

skipped-tests.tsv

Содержит информацию о пропущенных во время тестирования запросах, в том числе хранит причину, по которой данный запрос был пропущен.

slow-on-client.tsv

Содержит информацию о тестах, которые замедлились на клиенте. Существуют client time —суммарное время, потраченное на клиенте на запрос и server time - время, затраченное сервером. Если при тестировании запроса выявляется большая разница во времени между клиентским и серверным временами (такое бывает, если, например, клиент посылает слишком много данных), то данный запрос попадает в файл `slow-on-client.tsv`.

test-times.tsv

Содержит информацию о времени прохождения тестов в системе. Имеет следующие поля:

Wall clock time — это полное время выполнения одного теста. Туда входят все запросы из этого теста, создание таблиц, проверки на сервере и т.п.

Total client time – это суммарное время, потраченное на клиенте на выполнение для конкретного запроса. Это время включает себя работу на сервере, получение клиентом ответа по сети и его обработка.

Test – имя теста.

Total queries – общее число запросов, входящих в конкретный тест.

Ignored short queries – проигнорированные запросы.

Longest query sum for all runs – самый медленный по времени тестирования запрос в тесте.

Avg wall clock time (sum for all runs) – среднее время на запрос с учётом всей подготовки.

Shortest query (sum for all runs) – самый быстрый по времени тестирования запрос в тесте.

unstable-queries.tsv

Содержит информацию о нестабильных запросах, т.е. тех запросах, которые показывают сильную разницу во времени тестирования от запуска к запуску.

report-errors.rep

Содержит информацию об ошибках, которые возникли при обработке файлов.

all-query-runs.json

Файл json, который содержит информацию о времени всех запусков запросов. Так как запросы запускаются несколько раз, а затем берется среднее значение полученных времен, данный файл также содержит информацию о повторных запусках каждого запроса.

Для создания интерфейса в качестве входных данных были взяты отдельный коммит и набор из 15-ти предшествующих ему коммитов для отображения истории изменения данных тестирования. С помощью несложных запросов на bash описываемые выше необходимые файлы были скопированы в

отдельную папку для каждого коммита, результаты тестирования которого будут описываться в веб-приложении.

Стоит упомянуть, что до создания веб-приложения статическая html страница с результатами тестов генерировалась с помощью программы на языке Python – `report.py`. В ходе разработки был создан другой файл на том же языке, но генерирующий json для frontend-части приложения.

Данный файл считывает информацию из необходимых папок, обрабатывает, преобразует и складывает ее в файл json – отдельный для каждого коммита. После данной обработки json файлы складываются в папку рядом с файлами frontend-части, чтобы к ним можно было обратиться из кода.

2.2 Создание проекта и его структура

Чтобы создать проект, можно просто добавлять в папку необходимые файлы, а можно воспользоваться системами сборки, например Vue CLI [20]. При использовании выбранного фреймворка Nuxt.js уменьшается количество кода, необходимое для работы проекта, а также становится доступным функционал для создания статического приложения, который нам необходим. При этом для сборки проекта с помощью Nuxt.js необходимо строгое соблюдение названия корневых папок и некоторых файлов. Именно поэтому проще всего создать проект с помощью всего лишь одной простой команды, которая создаст для работы все возможные папки (о структуре проекта на Nuxt будет рассказано чуть ниже), стандартный файлы и начальные зависимости:

```
$ vue init nuxt-community/starter-template
<project-name>
```

Когда проект создан, необходимо зайти в него и установить зависимости командой:

```
$ npm install
```

Теперь можно запустить проект:

```
$ npm run dev
```

Готово.

Рассмотрим структуру проекта, немного подробнее останавливаясь на папках, которые использовались в работе. Ниже представлена структура проекта (не все папки из возможных при работе с Nuxt.js использовались).

```
clickhouse-testing-interface /
-- | assets /
-- | components /
-- | dist /
----- | _nuxt /
----- | test_data /
```

```
----- | 200.html
----- | index.html
-- | layouts /
-- | node_modules /
-- | pages /
-- | plugins /
-- | static /
----- | test_data /
-------- | commit /
----------- | json-to-send.json
-------- | commit_1 /
-------- | ...
-- | store /
-- | nuxt.config.js
-- | README.md
```

В проекте также присутствуют другие файлы конфигурации, которые важны для проекта, но не настроены вручную во время этой разработки, поэтому они будут просто упомянуты, так как они были оставлены без изменений по умолчанию.

```
-- | .editorconfig
-- | .eslintrc.js
-- | .gitignore
-- | package.json
-- | package-lock.json
```

Папка «assets» содержит нескомпилированные файлы с исходным кодом, такие как LESS, SASS, картинки или шрифты.

Папка «components» содержит компоненты Vue для проекта.

Папка «dist» генерируется автоматически после запуска сборки

```
$ npm run generate
```

Данная папка «dist» – необходимый нам бандл, который содержит в себе папку с кодом js, всеми статичными файлами, в том числе картинками, шрифтами и входными данными json. Помимо этого, генерируется файл «200.html», который отображается по умолчанию при открытии проекта или при переходе в проекте по несуществующему адресу. Такой файл страхует нас от того, что может быть показана совершенно неясная пользователю страница при ошибке перехода – всю информацию такого файла можно прописать самостоятельно.

Также при безошибочной сборке проекта появляется файл «index.html». Именно этот файл и будет открываться для отображения всего интерфейса. В нем не просто указаны зависимости от других файлов, в нем уже в виде текста прописан статичный html код, который не генерируется, а просто открывается.

Папка «layouts» содержит в себе шаблоны приложения. Данную папку нельзя переименовывать или удалять без определённого изменения файлов конфигурации, поэтому данная папка с единственным пустым по наполнению файлом осталась без изменений.

Папка «node_modules» генерируется после выполнения команды:

```
$ npm install
```

Данная папка «node_modules» содержит в себе все зависимости и библиотеки, необходимые для работы приложения. Без данной папки с необходимыми зависимостями проект нельзя будет запустить. Так как данная папка занимает существенный объем памяти, при этом содержит, зачастую, тысячи файлов, то она не выгружается на GitHub. Для примера на последних

этапах разработки данного веб-приложения данная папка занимала 230 МБ на диске и содержала 24 793 объекта. Frontend- код перемещается без папки «node_modules», а после при необходимости запуска проекта ее просто генерируют одной командой, о которой писалось выше.

Папка «pages» содержит представления (views) и маршруты (routes). Для каждого файла расширения .vue внутри данной папки автоматически генерируется маршрут до него внутри приложения. Данная отличительная особенность сборки проекта с помощью Nuxt значительно ускоряет работу над кодом, так как не возникает необходимость создавать файл для прописывания всех маршрутов внутри приложения, а также не возникает путаницы с сопоставлением имени страницы и пути до нее. Так, если внутри папки pages создать файл «about.vue», то будет автоматически сгенерирован путь до него «/about». По умолчанию главная страница по пути «/» отображает файл «index.vue». В данном проекте содержится только одна страница «index.vue».

Папка «plugins» содержит JavaScript-плагины, которые будут запущены перед сборкой корневого приложения Vue.js. Для текущего проекта использовался один по смыслу плагин, который был оформлен в файл «vue-highcharts.js». С помощью данного файла к проекту подключались библиотека для построения графиков Highcharts.js, а также расширение для нее (для построения дополнительных видов графиков) highcharts-more, которое содержится в highcharts, но подключается отдельно.

При подключении модулей может возникнуть ошибка, так как код при сборке загружается несколько раз, поэтому все объявления плагинов обертываются в такое условие:

```
if (typeof Highcharts === 'object') {
    <name-of-plugin-from-highcharts>(Highcharts);
}
```

Папка «static» содержит в себе статические файлы, которые доступны для обращения через /. В данном проекте в данной папке содержится информацию по каждому из визуализируемых коммитов. Как описано в структуре файлов выше, вся информация лежит по адресу «static/test_data/commit_<number-of-commit>/json-to-send.json». Информация по текущему главному коммиту содержится по адресу «static/test_data/commit/json-to-send.json».

Папка «store» содержит в себе данные о хранилище, о котором подробнее будет сказано в разделе «2.3 Хранилище store». Однако, стоит упомянуть, что хранилище при работе с Nuxt опять же строится по особым правилам. В проекте используется версия оформления хранилища, при которой каждый файл расширения .js превращается в самостоятельный модуль store. Это ускоряет работу над кодом, так как не создается отдельный файл, который собирает все модули вместе – вся синхронизация и сборка происходит автоматически, благодаря соблюдению правил оформления кода.

Файл «nuxt.config.js» конфигурационный, то есть содержит набор настроек для приложения, которые могут быть добавлены/удалены/изменены пользователем при необходимости.

В данном файле указана мета-информация, подключены файлы с плагинами и добавлены некоторые правила для сборки. В последней версии Nuxt на момент разработки 2.12.2 содержался баг, при котором в итоговый html

файл все файлы, в том числе скрипты, подключались по пути с лишним / в начале. Например, вместо пути src="_nuxt/9cda04d213a43688279a.js" конструктор прописывал путь src="/_nuxt/9cda04d213a43688279a.js", что не позволяло подключить файл. Данная проблема была решена с помощью добавления в файл «nuxt.config.js» следующего кода:

```
build: {
        extend(config, ctx) {
            if(!ctx.isDev) {
                config.output.publicPath = '_nuxt/'
            }
        }
},
```

Данный код указывает конкретное начало пути к файлам, что и было необходимо.

Файл README.md – это файл формата markdown, в котором пишется информация о проекте, правилах его сборки и прочая информация, которая может быть полезна пользователю. Весь текст автоматически отображается после структуры проекта на GitHub.

2.3 Хранилище store

Большинство веб-приложение используют хранилища, которое содержит необходимые для работы приложения данные. Рассмотрим преимущества использования хранилища по сравнению с хранением данных в компонентах.

Во-первых, одни и те же данные могут использоваться разными компонентами, поэтому неэффективно и сложно передавать от одних компонентов в другие данные локально, так как может возникнуть путаница. Намного удобнее было бы задать переменные глобально, и менять их из любого компонента. Такую возможность предоставляет хранилище store.

Во-вторых, при изменении одним компонентом данных, у остальных компонентов они автоматически обновятся. Помимо этого, store защищает от случайного изменения данных. Чтобы это понять, необходимо обратиться к структуре хранилища. В данном проекте файлы расширения .js, которые лежат в папке «store», автоматически становятся для системы сборки модулями. Как правило, хранилище делят на подобные модули, когда приложение использует большое количество данных, которые необходимо разделить по смыслу. Логичнее всего определять различные данные в разные модули для удобства работы и избегания нагромождения кода в одном файле. Так как все внешние данные о результатах тестирования, которые необходимы для работы приложения, тесно связаны по смыслу, их было решено положить в один модуль «tests.js»

Структуру модуля хранилища можно разделить на 4 части.

State – состояние, которое содержит все данные. Его нельзя изменить напрямую. Это сделано для простоты использования хранилища и правильной обработки получаемой им информации. Чтобы получить данные из state необходимо использовать getters (геттеры) – это набор функций, которые имеют доступ к хранилищу и могут передать необходимые данных. Они защищают от прямого доступа к данным приложения, передавая только необходимую информацию и не показывая лишнее. Чтобы изменить данные, а также чтобы выполнить запрос на backend часть, используются actions. Эти «действия» (по факту, функции) обрабатывают информацию, которую предает пользователь (например, имя, на которое нужно заменить переменную); если

используется сервер, то делают запрос на изменение данных на backend части приложения и после получения положительного ответа, обращаются к mutations для изменения данных в state. Mutations – это функции, которые имеют доступ к state и могут его изменить.

Наглядное представление о работе хранилища представлено на схеме (Рис. 2.1) с официальной документации Vuex – инструмента, с помощью которого хранилище можно использовать в приложениях, написанных на Vue.js.

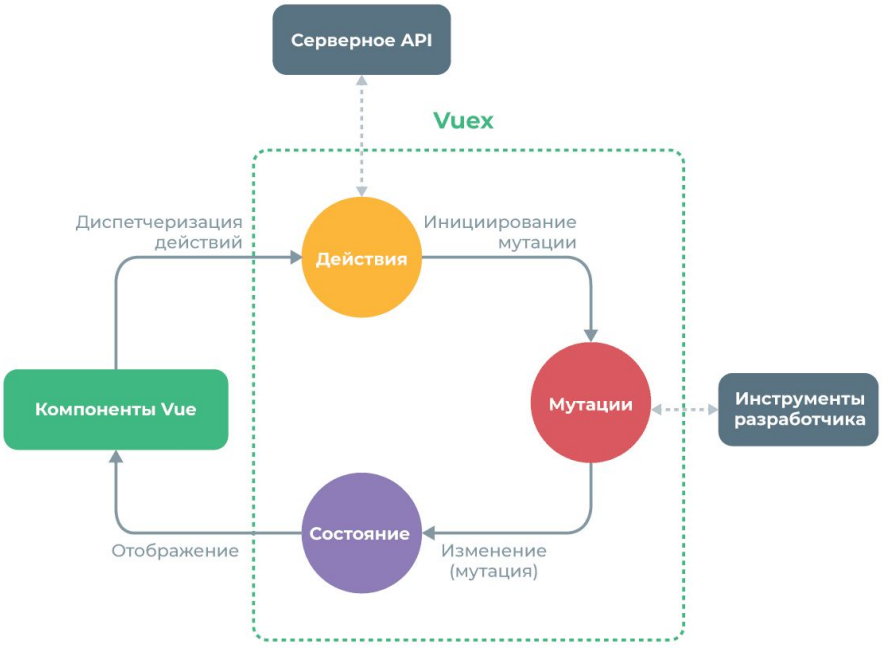


Рис. 2.1. Схема работы хранилища Vuex

Так как в разрабатываемом приложении отсутствует backend-часть, в хранилище данные попадают из файлов, которые находятся в папке static. Для простоты использования, информация о текущем коммите сохраняется в объекте store currentCommitInfo. Информация о других коммитах сохраняется в массив commitsHistory с подобными currentCommitInfo объектами. Так как на

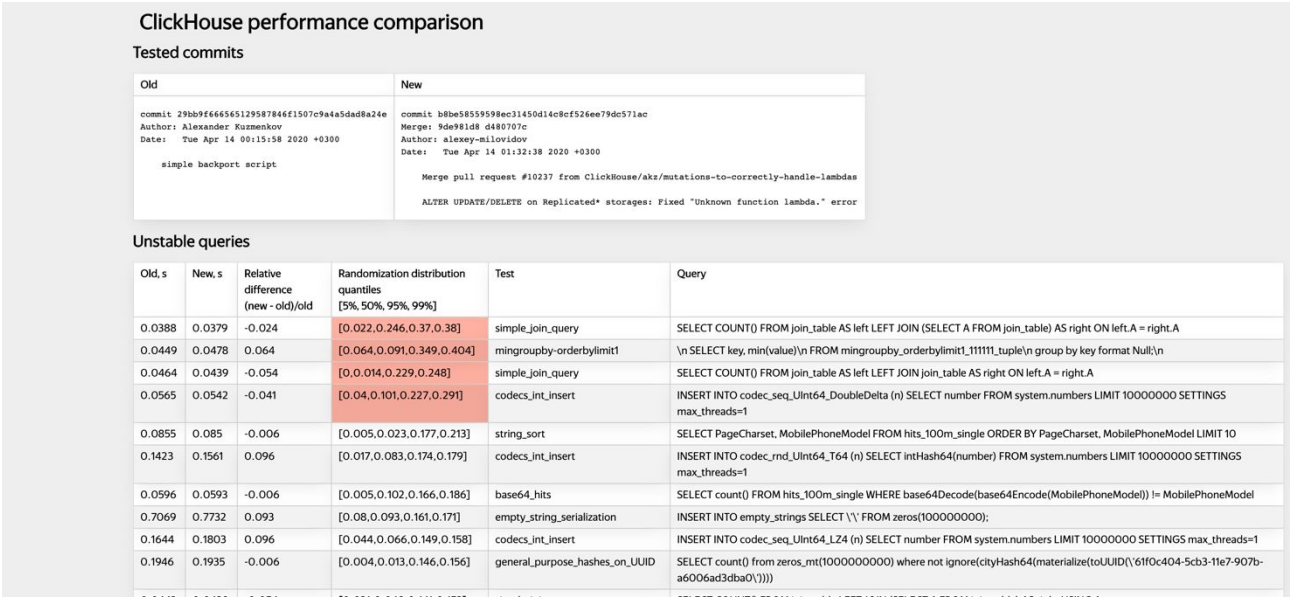
текущий момент разработки данный интерфейс лишь визуализирует данные, функции внутри actions и mutation отсутствуют, однако хранилище разработано так, чтобы при необходимости данный функционал можно было легко добавить.

2.3 Выводы и результаты

Для работы с данными была создана программа на языке Python3, которая отбирает необходимые файлы из общих результатов тестирования, считывает информацию из них, обрабатывает и представляет ее в удобном для веба формате json. Специально для сборки статического приложения с помощью фреймворка Nuxt.js было создано приложение со строгой иерархией файлов и папок. Для использования данного фреймворка были изучены правила написания кода и оформления структуры файлов, чтобы после сборки проект работал корректно. Сразу после создания структуры проекта было добавлено хранилище store для безопасного и удобного доступа всех компонентов к данным о результатах тестирования. В следующей главе будет подробно разобран следующий этап создания веб-приложения - разработка непосредственно самого интерфейса.

Глава 3 – Разработка интерфейса

При разработке интерфейса было необходимо отобразить максимальное количество информации, при этом сделать интерфейс лаконичным и понятным пользователю. Для создания интерфейса необходимо было изучить концепты дизайна интерфейса, в том числе решение, которое присутствует в ClickHouse на данный момент, и выяснить их сильные и слабые стороны. На Рисунке 3.1 приведена часть интерфейса, на котором раньше отображались результаты тестирования.



ClickHouse performance comparison

Tested commits

Old	New
commit 29bb9f6665651295878466f1507c9a4a5dad8a24e Author: Alexander Kuzmenkov Date: Tue Apr 14 00:15:58 2020 +0300 simple backport script	commit b8be58559598ec31450d14c8cf526ee79dc571ac Merge: 9de981d8 d480707c Author: alexey-milovidov Date: Tue Apr 14 01:32:38 2020 +0300 Merge pull request #10237 from ClickHouse/akuzm/mutations-to-correctly-handle-lambdas ALTER UPDATE/DELETE on Replicated* storages: Fixed "Unknown function lambda." error

Unstable queries

Old, s	New, s	Relative difference (new - old)/old	Randomization distribution quantiles [5%, 50%, 95%, 99%]	Test	Query
0.0388	0.0379	-0.024	[0.022,0.246,0.37,0.38]	simple_join_query	SELECT COUNT() FROM join_table AS left LEFT JOIN (SELECT A FROM join_table) AS right ON left.A = right.A
0.0449	0.0478	0.064	[0.064,0.091,0.349,0.404]	mingroupby-orderbylimit1	\n SELECT key, min(value)\n FROM mingroupby_orderbylimit1_111111_tuple\n group by key format Null;\n
0.0464	0.0439	-0.054	[0,0.014,0.229,0.248]	simple_join_query	SELECT COUNT() FROM join_table AS left LEFT JOIN join_table AS right ON left.A = right.A
0.0565	0.0542	-0.041	[0.04,0.101,0.227,0.291]	codecs_int_insert	INSERT INTO codec_seq_UInt64_DoubleDelta (n) SELECT number FROM system.numbers LIMIT 10000000 SETTINGS max_threads=1
0.0855	0.085	-0.006	[0.005,0.023,0.177,0.213]	string_sort	SELECT PageCharset, MobilePhoneModel FROM hits_100m_single ORDER BY PageCharset, MobilePhoneModel LIMIT 10
0.1423	0.1561	0.096	[0.017,0.083,0.174,0.179]	codecs_int_insert	INSERT INTO codec_rnd_UInt64_T64 (n) SELECT intHash64(number) FROM system.numbers LIMIT 10000000 SETTINGS max_threads=1
0.0596	0.0593	-0.006	[0.005,0.102,0.166,0.186]	base64_hits	SELECT count() FROM hits_100m_single WHERE base64Decode(base64Encode(MobilePhoneModel)) != MobilePhoneModel
0.7069	0.7732	0.093	[0.08,0.093,0.161,0.171]	empty_string_serialization	INSERT INTO empty_strings SELECT \'\' FROM zeros(100000000);
0.1644	0.1803	0.096	[0.044,0.066,0.149,0.158]	codecs_int_insert	INSERT INTO codec_seq_UInt64_LZ4 (n) SELECT number FROM system.numbers LIMIT 10000000 SETTINGS max_threads=1
0.1946	0.1935	-0.006	[0.004,0.013,0.146,0.156]	general_purpose_hashes_on_UUID	SELECT count() from zeros_mt(1000000000) where not ignore(cityHash64(materialize(toUUID(\'61f0c404-5cb3-11e7-907b-a6006ad3dba0\'))))

Рис. 3.1. Интерфейс, на котором раньше отображались результаты тестирования.

Данный интерфейс имел следующие недостатки: в нем отсутствовали графики, таблицы не имели сортировку, а чтобы просмотреть логи, все запросы и скачать архив со всеми результатами тестирования, необходимо было пролистывать всю страницу до самого конца.

После анализа существующих концептов дизайна интерфейса, наиболее подходящим был выбран вариант оформления в стиле dashboard как наиболее подходящий в данном случае. В подобном концепте слева располагается меню для переключения на разделы сайта, а в центре отображаются различные компоненты. При этом при смене раздела страница не перезагружалась, а изменяла отображаемый компонент в центре. В итоге, получился минималистичный интерфейс, который устранил главные недостатки предыдущего решения.

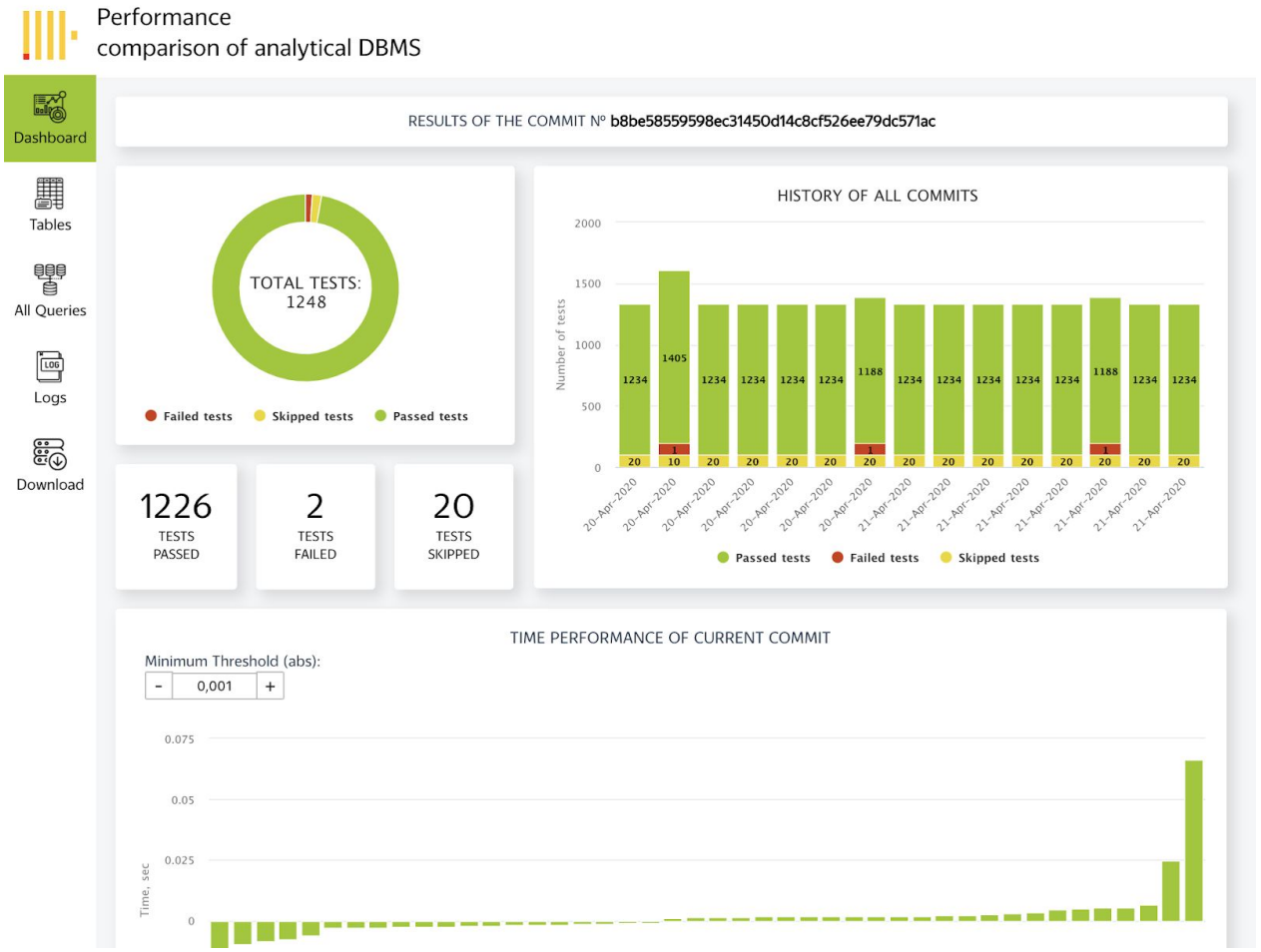


Рис. 3.2. Готовый разработанный интерфейс для визуализации ревизий ClickHouse. Часть раздела Dashboard.

Теперь же в интерфейсе присутствуют графики, а сделанные с помощью визуальной составляющей акценты обращают внимание пользователя на важные детали (Рис. 3.2). Так, например, количество пройденных/упавших/пропущенных тестов выведены в отдельные блоки в начале страницы. С помощью бокового меню решалась проблема быстрого получения доступа к таблицам, списку всех запросов при тестировании, логам после запуска кода и скачиванию архива с полными данными о тестировании.

Графики и диаграммы нарисованы при помощи библиотеки Highcharts, но остальной интерфейс был полностью реализован без использования посторонних библиотек – использовались лишь HTML5, CSS3 и Vue для быстрой передачи данных в компоненты и отображения однотипных данных

(например, строк в таблице). При необходимости на страницу можно добавлять неограниченное количество визуальных элементов, но необходимо было выбрать самые важные данные для быстро понимания результатов тестирования.

Рассмотрим каждый разделе интерфейса по отдельности.

3.1 Dashboard

При открытии приложения страница Dashboard, которая содержит визуализированную с помощью графиков и CSS, показывается автоматически (Рис. 3.3).

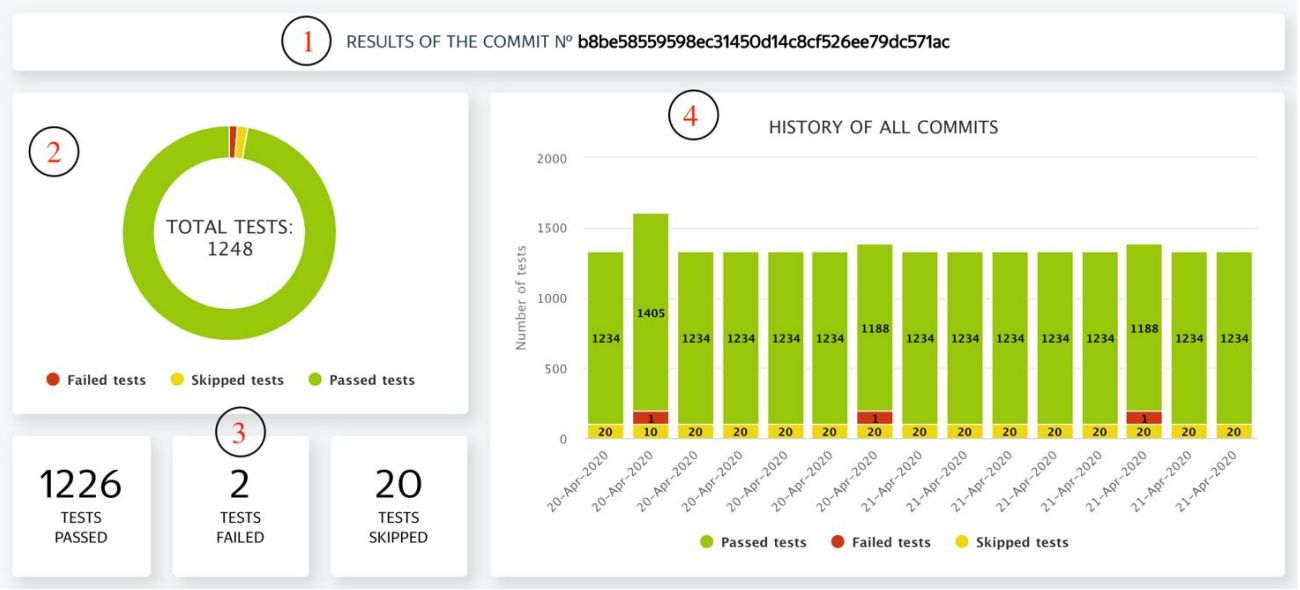


Рис. 3.3. Часть раздела Dashboard.

В самом начале страницы (цифра 1 на Рисунке 3.3) мы видим заголовок с названием текущего коммита, при клике на который пользователь переходит на страницу GitHub, на которой показываются изменения в коде этого коммита.

Чуть ниже (цифра 2 на Рисунке 3.3) видим круговую диаграмму, в которой разными цветами отображено соотношение количества

пройденных/упавших/пропущенных запросов при тестировании. В центре диаграммы указано общее количество тестов (=запросов) – так наглядно с помощью простого добавления текста внутри визуального компонента показывается, что все виды запросов в сумме дают полный набор тестов. При этом, восприятие целостности диаграммы и понимание соотношения не пропадает.

При наведении на части диаграммы, показывается дополнительная информация, а также показывается анимация, встроенная в библиотеку Highcharts (Рис. 3.4). Подобное отображение пояснений к частям графиков, а также анимация при наведении присутствует во всех графиках в проекте, поэтому далее данное введение не будет поясняться в тексте.

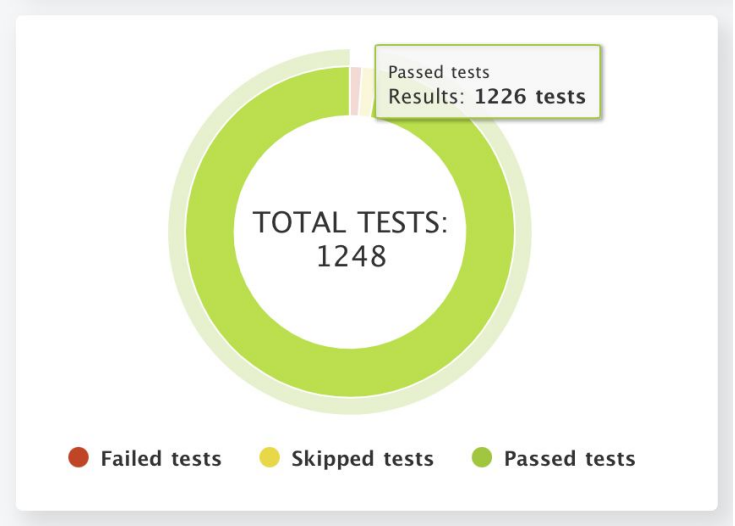


Рис. 3.4. Круговая диаграмма в разделе Dashboard.

Под цифрой 3 на Рисунке 3.3 отображено количество пройденных/упавших/пропущенных для их выделения среди остальной информации как важной.

Под цифрой 4 на Рисунке 3.3 отображен график соотношение количества пройденных/упавших/пропущенных запросов при тестировании 15-ти коммитов, которые далее будут называться «историческими данными». На

графике наглядно показано, в каких версиях кода были упавшие тесты и в какой день данные коммиты были залиты на GitHub.

Далее на странице Dashboard находятся три графика.

Первый график (Рис. 3.5) отображает среднее арифметическое разницы между новым и старым временем тестирования каждого из запросов внутри одного теста. Здесь под тестом подразумевается набор запросов с одинаковым testName. По оси Ox отображаются названия тестов. Данный график имеет настраиваемый параметр - Minimum Threshold (Рис. 3.6). Это минимальный порог, минимальное время по модулю, чтобы отобразить только тесты, которые ускорились или замедлились на определенное значение секунд, равное данному порогу.

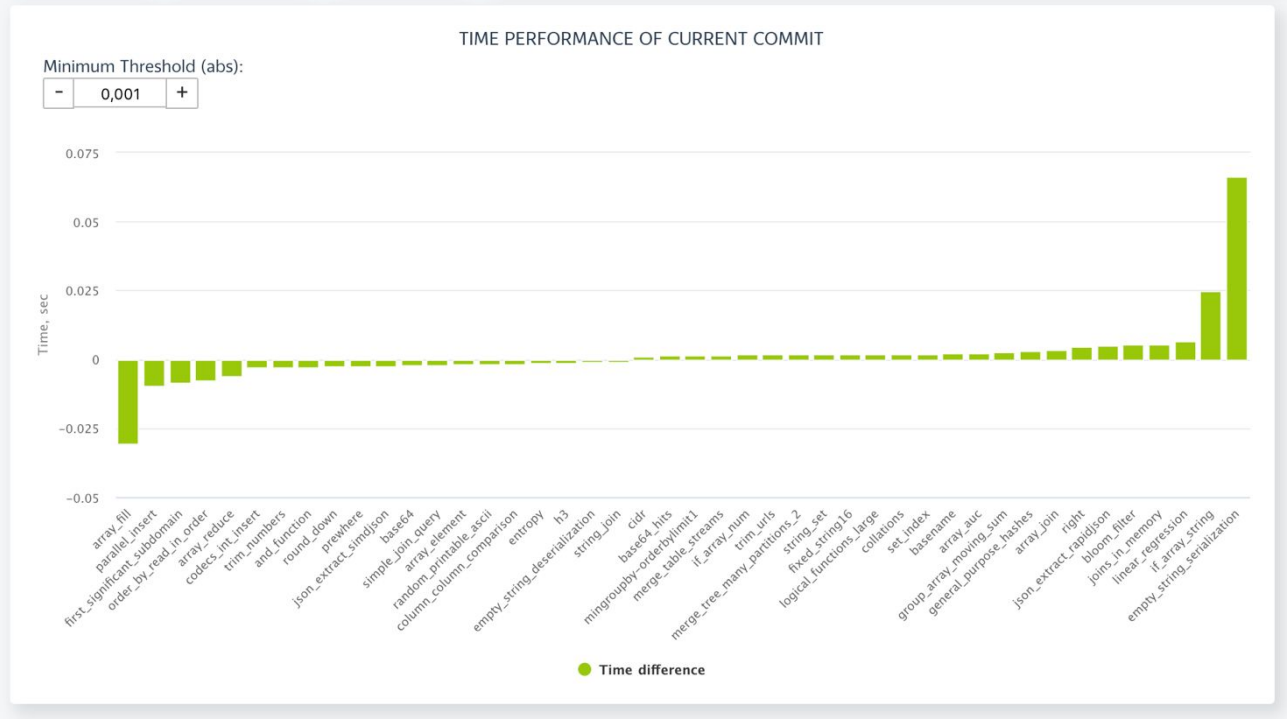


Рис. 3.5. Столбчатая диаграмма отображения ускорившихся и замедлившихся тестов с запросами. Минимальный порог равен 0.001 сек.

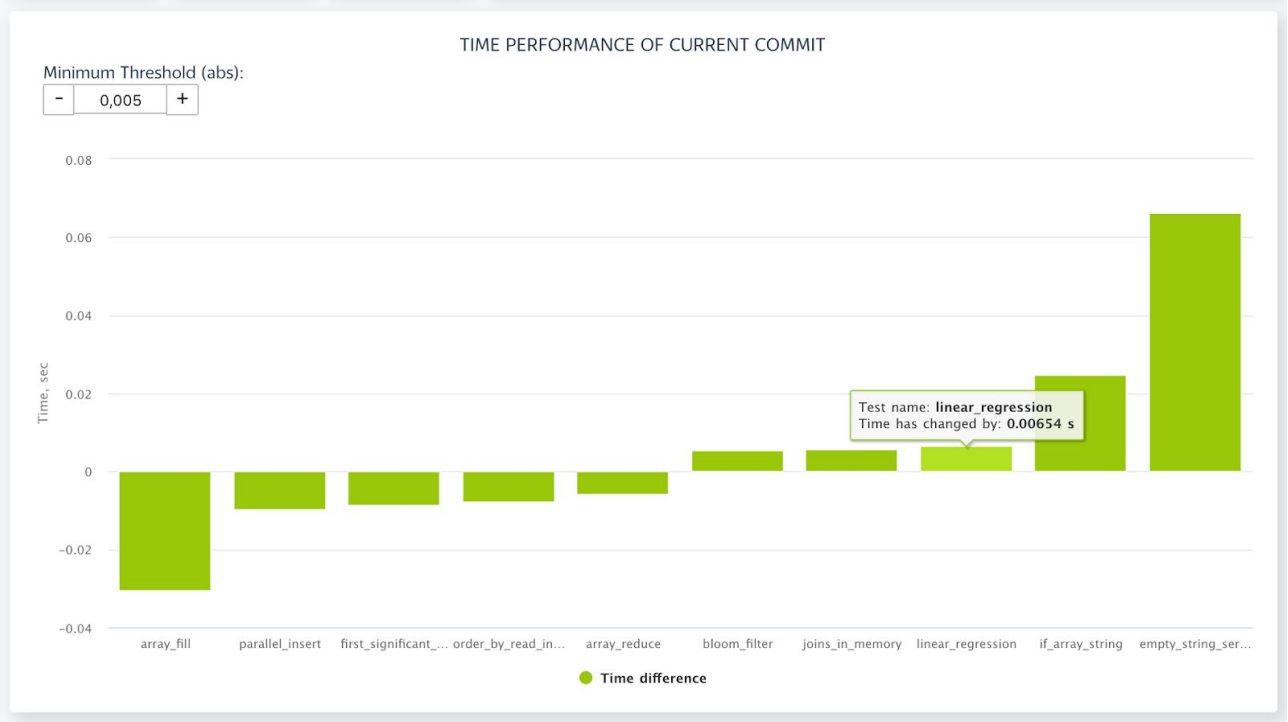


Рис. 3.6. Столбчатая диаграмма отображения ускорившихся и замедлившихся тестов с запросами. Минимальный порог равен 0.005 сек.

Следующий график (Рис. 3.7) показывает нестабильные запросы с доверительным интервалом для каждого из них. При этом здесь тоже присутствует параметр для подсчета данного интервала в виде перцентиля из массива [5%, 50%, 95%, 99%]. Каждый запрос при тестировании запускается несколько раз и при этом не всегда работает одно и то же время. Используя данные из «all-query-runs.json» о времени выполнения запроса строится доверительный интервал по формуле:

$$\bar{x} - Z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}} \leq \mu \leq \bar{x} - Z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}$$

Где $\bar{x}$ – это среднее время всех запусков запроса,

$Z_{\frac{\alpha}{2}}$ – константа для $\frac{\alpha}{2}$ - квантиля распределения,

$\frac{\sigma}{\sqrt{n}}$ – отношение дисперсии к корню количества запусков тестирования запроса.

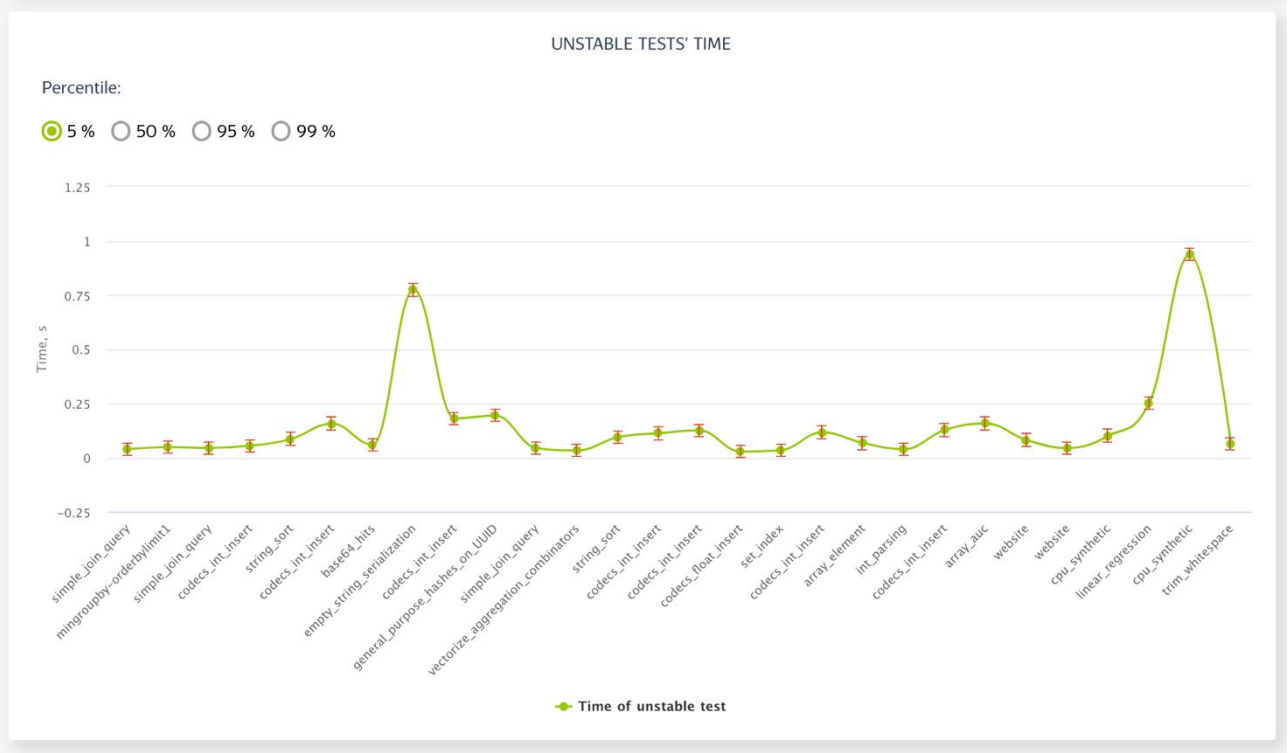


Рис. 3.7. График среднего времени тестирования нестабильных запросов с доверительным интервалом. Перцентиль равен 5%.

Третий график (Рис. 3.8), отображаемый на странице Dashboard – это график сравнения времени тестирования набора запросов, объединенных в один тест, со средним значением времени такого же набора, подсчитанным на «исторических данных». По оси Ox указаны названия наборов запросов (тестов).

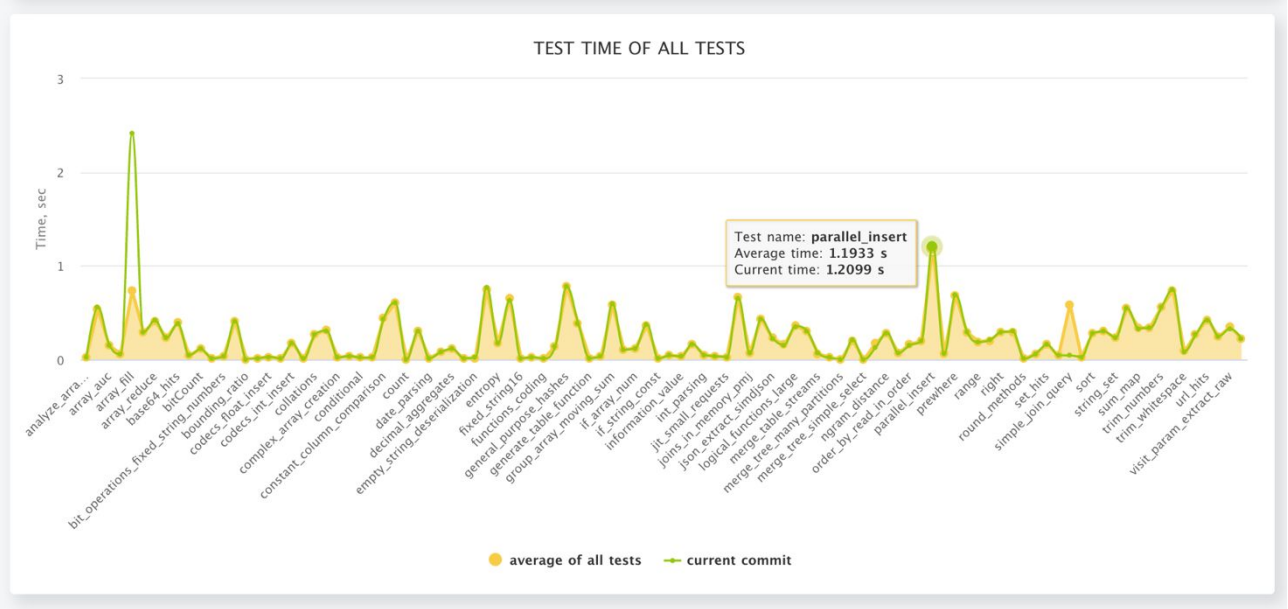


Рис. 3.8. График сравнения времени тестирования набора запросов, объединенных в один тест, со средним значением времени такого же набора

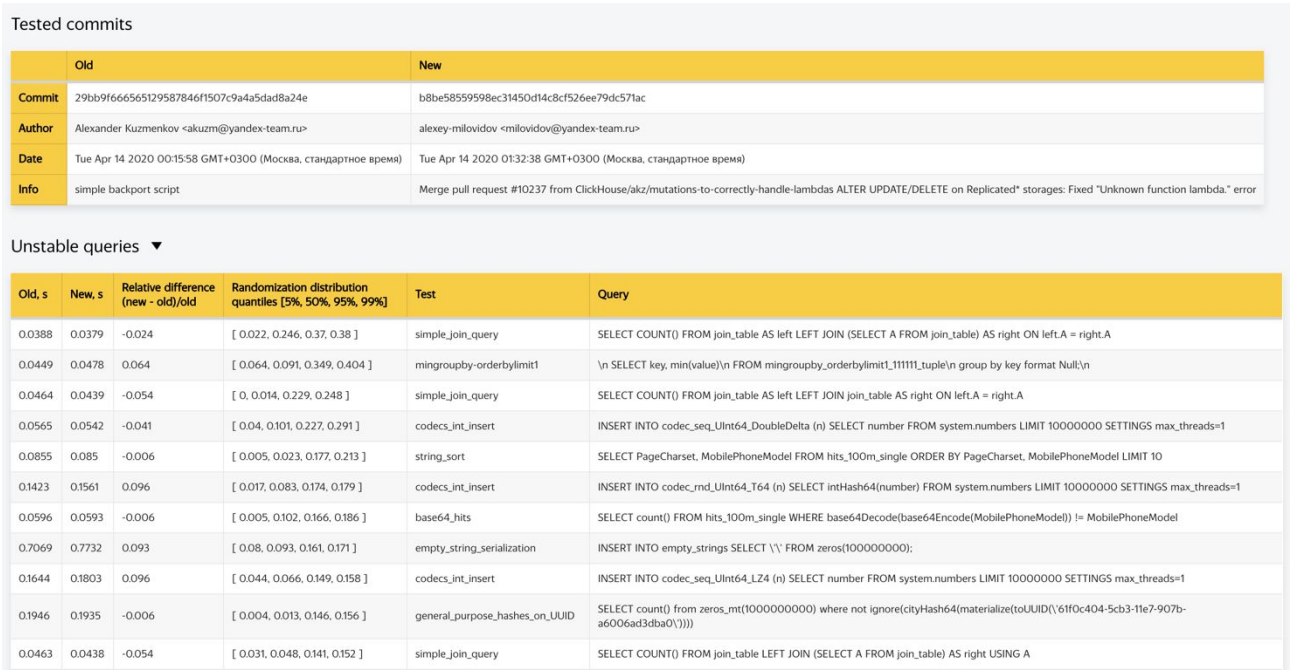
3.2 Таблицы и загрузчик всех файлов

В новом интерфейсе также необходимо было повторить таблицы, которые присутствовали в старой версии визуализации результатов. Однако, теперь они поддерживают сортировку. При клике на заголовок столбца он сортируется сначала в порядке убывания значений, если столбец содержит числа, в лексикографическом порядке, если это строка, и в порядке убывания элементов, начиная с первого, если это массив. При повторном клике на заголовок происходит так же сортировка, но уже в обратном порядке.

Все таблицы отображают данные, которые содержатся в файлах, о которых говорилось в разделе «2.1 Работа с файлами». Дополнительно к ним были добавлены стили, чтобы они сочетались с разработанным интерфейсом (Рис. 3.9).

Для удобства просмотра всех таблиц была добавлена функция их свертывания при нажатии специального значка справа от названия таблицы

(Рис. 3.10). При первом нажатии таблица скрывается, при повторном нажатии вновь показывается.



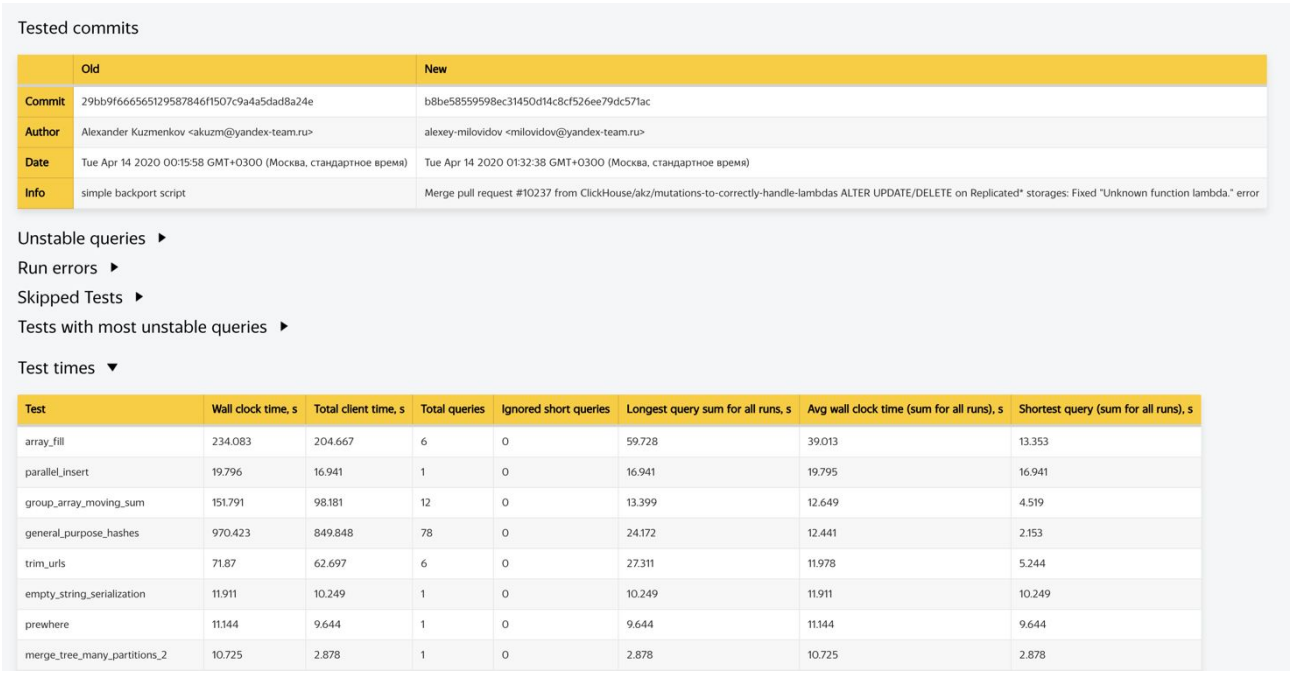
Tested commits

	Old	New
Commit	29bb9f666565129587846f1507c9a4a5dad8a24e	b8be58559598ec31450d14c8cf526ee79dc571ac
Author	Alexander Kuzmenkov <akuzm@yandex-team.ru>	alexey-milovidov <milovidov@yandex-team.ru>
Date	Tue Apr 14 2020 00:15:58 GMT+0300 (Москва, стандартное время)	Tue Apr 14 2020 01:32:38 GMT+0300 (Москва, стандартное время)
Info	simple backport script	Merge pull request #10237 from ClickHouse/akz/mutations-to-correctly-handle-lambdas ALTER UPDATE/DELETE on Replicated* storages: Fixed "Unknown function lambda." error

Unstable queries ▼

Old, s	New, s	Relative difference (new - old)/old	Randomization distribution quantiles [5%, 50%, 95%, 99%]	Test	Query
0.0388	0.0379	-0.024	[0.022, 0.246, 0.37, 0.38]	simple_join_query	SELECT COUNT() FROM join_table AS left LEFT JOIN (SELECT A FROM join_table) AS right ON left.A = right.A
0.0449	0.0478	0.064	[0.064, 0.091, 0.349, 0.404]	mingroupby-orderbylimit1	\n SELECT key, min(value)\n FROM mingroupby_orderbylimit1_111111_tuple\n group by key format Null;\n
0.0464	0.0439	-0.054	[0, 0.014, 0.229, 0.248]	simple_join_query	SELECT COUNT() FROM join_table AS left LEFT JOIN join_table AS right ON left.A = right.A
0.0565	0.0542	-0.041	[0.04, 0.101, 0.227, 0.291]	codecs_int_insert	INSERT INTO codec_seq_UInt64_DoubleDelta (n) SELECT number FROM system.numbers LIMIT 10000000 SETTINGS max_threads=1
0.0855	0.085	-0.006	[0.005, 0.023, 0.177, 0.213]	string_sort	SELECT PageCharset, MobilePhoneModel FROM hits_100m_single ORDER BY PageCharset, MobilePhoneModel LIMIT 10
0.1423	0.1561	0.096	[0.017, 0.083, 0.174, 0.179]	codecs_int_insert	INSERT INTO codec_rnd_UInt64_T64 (n) SELECT intHash64(number) FROM system.numbers LIMIT 10000000 SETTINGS max_threads=1
0.0596	0.0593	-0.006	[0.005, 0.102, 0.166, 0.186]	base64_hits	SELECT count() FROM hits_100m_single WHERE base64Decode(base64Encode(MobilePhoneModel)) != MobilePhoneModel
0.7069	0.7732	0.093	[0.08, 0.093, 0.161, 0.171]	empty_string_serialization	INSERT INTO empty_strings SELECT \'\' FROM zeros(100000000);
0.1644	0.1803	0.096	[0.044, 0.066, 0.149, 0.158]	codecs_int_insert	INSERT INTO codec_seq_UInt64_LZ4 (n) SELECT number FROM system.numbers LIMIT 10000000 SETTINGS max_threads=1
0.1946	0.1935	-0.006	[0.004, 0.013, 0.146, 0.156]	general_purpose_hashes_on_UUID	SELECT count() from zeros_mt(1000000000) where not ignore(cityHash64(materialize(toUUID(\'61f0c404-5cb3-11e7-907b-a6006ad3dba0\'))))
0.0463	0.0438	-0.054	[0.031, 0.048, 0.141, 0.152]	simple_join_query	SELECT COUNT() FROM join_table LEFT JOIN (SELECT A FROM join_table) AS right USING A

Рис. 3.9. Часть страницы, на которой в таблицах отображены данных о результатах тестирования из файлов раздела «2.1 Работа с файлами».



Tested commits

	Old	New
Commit	29bb9f666565129587846f1507c9a4a5dad8a24e	b8be58559598ec31450d14c8cf526ee79dc571ac
Author	Alexander Kuzmenkov <akuzm@yandex-team.ru>	alexey-milovidov <milovidov@yandex-team.ru>
Date	Tue Apr 14 2020 00:15:58 GMT+0300 (Москва, стандартное время)	Tue Apr 14 2020 01:32:38 GMT+0300 (Москва, стандартное время)
Info	simple backport script	Merge pull request #10237 from ClickHouse/akz/mutations-to-correctly-handle-lambdas ALTER UPDATE/DELETE on Replicated* storages: Fixed "Unknown function lambda." error

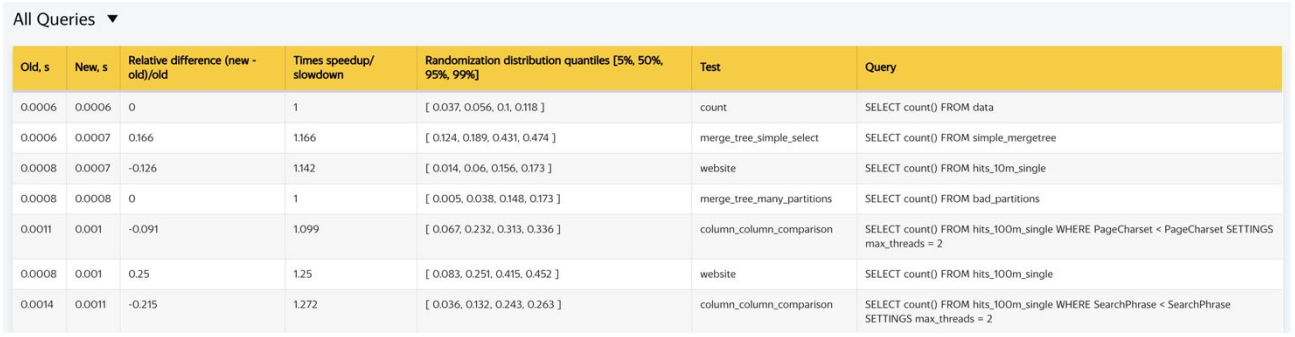
Unstable queries ▸
Run errors ▸
Skipped Tests ▸
Tests with most unstable queries ▸

Test times ▼

Test	Wall clock time, s	Total client time, s	Total queries	Ignored short queries	Longest query sum for all runs, s	Avg wall clock time (sum for all runs), s	Shortest query (sum for all runs), s
array_fill	234.083	204.667	6	0	59.728	39.013	13.353
parallel_insert	19.796	16.941	1	0	16.941	19.795	16.941
group_array_moving_sum	151.791	98.181	12	0	13.399	12.649	4.519
general_purpose_hashes	970.423	849.848	78	0	24.172	12.441	2.153
trim_urls	71.87	62.697	6	0	27.311	11.978	5.244
empty_string_serialization	11.911	10.249	1	0	10.249	11.911	10.249
prewhere	11.144	9.644	1	0	9.644	11.144	9.644
merge_tree_many_partitions_2	10.725	2.878	1	0	2.878	10.725	2.878

Рис. 3.10. Демонстрация свернутых и отображаемых таблиц в разделе «Tables».

Отдельно от всех остальных таблиц отрисовывается аналогичная таблица со всем запросами, завершившимися успешно (Рис. 3.11). Данная таблица отнесена в отдельный раздел интерфейса «All Queries», в связи ее смысловым различием с другими таблицами, а также из-за своего большого объема, превышающего объем всех таблиц, раздела «Tables» в несколько раз. Данные берутся из обработанного файла all-queries.tsv.

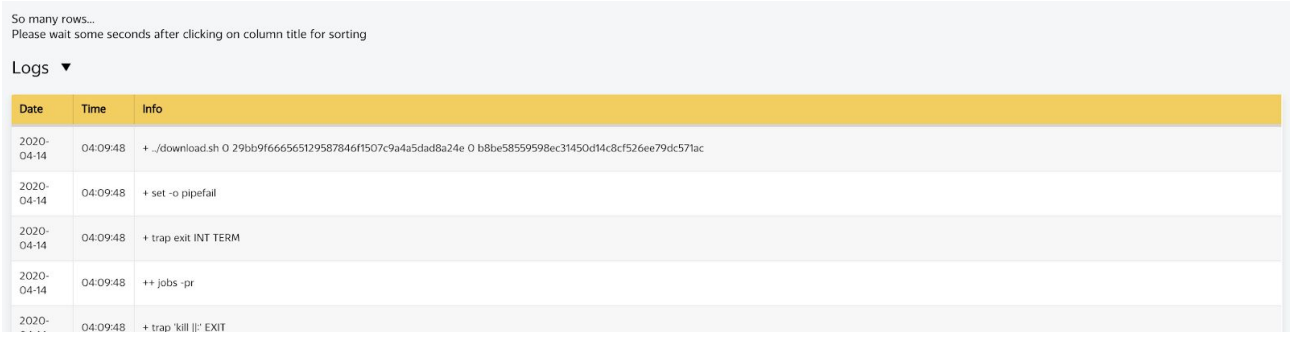


All Queries ▼

Old, s	New, s	Relative difference (new - old)/old	Times speedup/ slowdown	Randomization distribution quantiles [5%, 50%, 95%, 99%]	Test	Query
0.0006	0.0006	0	1	[0.037, 0.056, 0.1, 0.118]	count	SELECT count() FROM data
0.0006	0.0007	0.166	1.166	[0.124, 0.189, 0.431, 0.474]	merge_tree_simple_select	SELECT count() FROM simple_mergetree
0.0008	0.0007	-0.126	1.142	[0.014, 0.06, 0.156, 0.173]	website	SELECT count() FROM hits_10m_single
0.0008	0.0008	0	1	[0.005, 0.038, 0.148, 0.173]	merge_tree_many_partitions	SELECT count() FROM bad_partitions
0.0011	0.001	-0.091	1.099	[0.067, 0.232, 0.313, 0.336]	column_column_comparison	SELECT count() FROM hits_100m_single WHERE PageCharset < PageCharset SETTINGS max_threads = 2
0.0008	0.001	0.25	1.25	[0.083, 0.251, 0.415, 0.452]	website	SELECT count() FROM hits_100m_single
0.0014	0.0011	-0.215	1.272	[0.036, 0.132, 0.243, 0.263]	column_column_comparison	SELECT count() FROM hits_100m_single WHERE SearchPhrase < SearchPhrase SETTINGS max_threads = 2

Рис. 3.11. Таблица всех запросов, завершившихся успешно, в разделе «All Queries».

В следующем разделе интерфейса отображается не менее большая по объему таблица с логами – отчетом о результате работы кода (Рис. 3.12). Данные берутся из файла обработанного файла compare.log.



So many rows...
Please wait some seconds after clicking on column title for sorting

Logs ▼

Date	Time	Info
2020-04-14	04:09:48	+ ../download.sh 0 29bb9f666565129587846f1507c9a4a5dad8a24e 0 b8be58559598ec31450d14c8cf526ee79dc571ac
2020-04-14	04:09:48	+ set -o pipefail
2020-04-14	04:09:48	+ trap exit INT TERM
2020-04-14	04:09:48	++ jobs -pr
2020-	04:09:48	+ trap 'kill ||:' EXIT

Рис. 3.12. Таблица всех запросов, завершившихся успешно, в разделе «All Queries».

Наконец, в последнем разделе «Downloads» или «Загрузки» содержится единственный компонент - кнопка, при нажатии на которую начинается загрузка архива со всеми результатами тестирования (в том числе тех файлов, которые не использовались для визуализации). Данный компонент ссылается на архив «output.7z», который лежит рядом с проектом, и после нажатия на кнопку начинается загрузка.

3.3 Выводы и результаты

Для того, чтобы пользователь мог быстро понять результаты тестирования, был разработан интерфейс, который при удобном и минималистичном дизайне предоставлял большое количество данных с результатами работы. Для создания визуализации использовались графики, диаграммы и таблицы, которые поддерживают сортировку данных по столбцам. Каждый график или диаграмма имеет анимацию для выделения какой-либо части рисунка и лучшего восприятия. Весь интерфейс генерируется динамически, поэтому при изменение входных данных, вся изменившаяся информация во всех компонентах автоматически обновляется.

Сборка проекта и публикация на GitHub

После завершения работы над предобработкой данных и интерфейсом, необходимо собрать проект. Данное веб-приложение должно работать без сервера и лежать вместе с результатами тестирования в репозитории ClickHouse на GitHub. Для сборки подобного приложения нам необходима сборка статичной страницы с помощью фреймворка Nuxt.js. Если говорить о самых распространенных вариантах рендеринга страниц, можно выделить 3 вида:

● Серверный рендеринг (Server-Side Rendering) – рендеринг на стороне сервера. После посылки запроса клиентом, на сервере генерируется весь HTML-код страницы, который затем посылается клиенту. Это очень сильно экономит ресурсы пользователя, так как ему приходит уже собранная html страница.

● Клиентский рендеринг (Client-Side Rendering) – рендеринг на стороне клиента. Сборка происходит прямо в браузере пользователя с помощью JavaScript. Шаблоны, маршруты, логика данных – все это обрабатывается на клиенте, а не на сервере.

● Статический рендеринг (Static Rendering) - рендеринг, при котором на запрос клиента сервер сразу же отправляет заранее сгенерированную статичную HTML- страницу. В отличии от серверного рендеринга, здесь генерация страницы происходит 1 раз задолго до посылаемого запроса.

Также существуют сложные виды рендеринга, в том числе комбинирующие описанные выше виды. Однако, для текущего веб-приложения подходит лишь один вид рендеринга - статический. Для того, чтобы запустить сборку, необходимо выполнить следующую команду:

```
$ npm run generate
```

При указании верных параметров для сборки в конфигурационных файлах, после завершения сборки в корне проекта появится папка dist, в которой и будет лежать все необходимое для расположения проекта на статическом холдинге.

Для работы данного проекта, его необходимо добавить в код проекта ClickHouse. Получается, что после добавления кода рядом с результатами

тестирования, к которым будет иметься доступ, статичная страница будет генерироваться по следующему сценарию.

1. Сразу после завершения тестирования и формирования файлов с его результатами запускается Python файл, отвечающий за выбор необходимых данных, их обработку и генерацию json-ов для frontend-части проекта.

2. Код интерфейса получает доступ к новым результатам, отраженных в формате json. С помощью специально написанного скрипта запускается сборка новой версии статичной страницы, после чего получается папка dist со всеми необходимыми данными для запуска статичной HTML страницы.

3. Папка dist располагается рядом со старой статичной HTML страницей, так как первое время проект проработает в тестирующем режиме для добавления функционала и устранения возможных ошибок. Именно поэтому у пользователя будет возможность просматривать как старый вариант интерфейса для визуализации, так и новую версию.

Весь код frontend-части веб-приложения сохранен в 2 видах: версия с использованием Nuxt.js для генерации статичной страницы HTML и версия без Nuxt.js для последующей сборки кода с помощью другого сборщика и публикации на сервере.

Результаты проделанной работы можно увидеть по ссылкам в репозиториях GitHub:

1. Версия со сборкой Nuxt.js:
   https://github.com/DariaHighfly/clickhouse-testing-interface
2. Версия без Nuxt.js:
   https://github.com/DariaHighfly/clickhouse-revision-interface

Заключение

В ходе работы были исследованы существующие решения, способы визуализации данных в сети Интернет и представление результатов тестирования в целом. Помимо этого, были подробно изучены необходимые материалы для разработки веб-интерфейса– существующие языки программирования, фреймворки и библиотеки. Для разработки было важно создать статическое веб-приложение, которое могло было быть расположено на статическом ресурсе, в данном случае в репозитории основанного кода open-source проекта ClickHouse. Для подобной сборки были изучены несколько технологий, среди которых был отобран фреймворк Nuxt.js в связи с его эффективной работой и быстрой интеграцией с кодом на Vue.js – основным фреймворком, выбранным для разработки данного проекта.

В результате работы была создана программа на языке Python3, которая отбирает необходимые файлы из общих результатов тестирования, считывает результаты тестирования из них, обрабатывает и представляет полученные данные в удобном для веба формате json. Данный формат также удобен для работы с JavaScript – главным языком для веб-разработки frontend-части приложений. Помимо обработки большого количества данных с результатами тестирования ревизий кода ClickHouse, был разработан интерфейс сравнения и анализа производительности данных ревизий.

Для работы данного интерфейса были использованы язык программирования JavaScript, языки разметки и стилей HTML5 и CSS3, фреймворки Vue.js и Nuxt.js, а также библиотеки Vuex и Highcharts.js для работы с хранилищем и построения графиков и диаграмм соответственно. При создании визуализации в виде набора графиков, диаграмм и таблиц, а также самого интерфейса для удобного просмотра отображаемой информации, были совмещены идеи показа наибольшего количества информации о тестировании

ревизий и удобного и лаконичного дизайна страницы. Для достижения данной цели на графиках была показана самая важная информации о результатах тестирования, представленная в удобной форме, большой объем данных о запросах и логах работы программы были представлены в виде сортируемых таблиц. Весь интерфейс генерируется динамически, а значит при изменение входных данных, например, изменения некоторых запросов и их числа, данная информация после генерации статичной HTML страницу автоматически обновится в тех местах, где она отображалась. Это позволяет использовать данный код интерфейса для отображения результатов тестирования разных ревизий, а также для работы вместе с сервером, который может посылать данные о конкретных коммитах по запросу.

Все полученные данные сохраняются в созданном с помощью Vuex внутри кода хранилище store, откуда разные компоненты приложения имеют безопасный доступ к хранящейся информации о результатах тестирования.

На данный момент проект внедрен в код ClickHouse в тестовом режиме для добавления нового функционала и устранения возможных ошибок. В дальнейшем данный интерфейс может быть легко дополнен другими визуальными элементами. При изменении структуры входных данных код проекта может быть быстро изменен в связи с удобным разделением кода на компоненты. Так, может быть исправлена только небольшая часть кода, которая не повлияет на работу остальных компонентов, поскольку почти каждый компонент работает самостоятельно.

В итоге, при реализации данной дипломной работы были совмещены идеи обработки и анализа данных и использование веб-технологий для их визуализации. Помимо работы с подготовкой данных с помощью языка Python, был полностью проделан путь разработки полноценного статического веб-приложения с нуля. В будущем было бы интересно продолжить развитие темы отображения данных в вебе так, чтобы информация воспринималась

пользователем максимально легко и быстро. Возможно, именно на примере развития данного веб-приложения получится распространить идею важности правильного представления тестирующих данных в web для их быстрого восприятия и понимания пользователем.

Список литературы

[1] Oracle. Документация // Oracle.com [Электронный ресурс]. URL: https://www.oracle.com/ru/database/what-is-database.html (дата обращения: 20.05.2020).

[2] ClickHouse Документация ClickHouse // Clickhouse.tech [Электронный ресурс]. URL: https://clickhouse.tech/docs/ru/ (дата обращения: 20.05.2020).

[3] NuxtJS [Электронный ресурс]. URL: https://nuxtjs.org/guide#features (дата обращения: 20.05.2020).

[4] Веб-документация MDN [Электронный ресурс]. URL: https://developer.mozilla.org/ru/docs/Web/Web_Components (дата обращения: 20.05.2020).

[5] Веб-документация JavaScript // learn.javascript.ru [Электронный ресурс]. URL: https://learn.javascript.ru/web-components (дата обращения: 20.05.2020).

[6] Suren Machiraju, Suraj Gaurav, Apress L.P DevOps for Azure applications : deploy web application on Azure / Suren Machiraju, Suraj Gaurav, Apress L.P, New York Apress, 2018.

[7] Volodymyr Melymuka TeamCity 7 continuous integration essentials / Volodymyr Melymuka, Birmingham: Packt Pub, 2012.

[8] Moutsatsos I.K. [и др.]. Jenkins-CI, an Open-Source Continuous Integration System, as a Scientific Data and Image-Processing Platform // SLAS DISCOVERY: Advancing the Science of Drug Discovery. 2016. № 3 (22). С. 238–249.

[9] Maronna R. Charu C. Aggarwal and Chandan K. Reddy (eds.): Data clustering: algorithms and applications // Statistical Papers. 2015. № 2 (57). С. 565–566.

[10] Alghasali S., Kuznetsova A., Aivazov V. FEATURES OF CLUSTERING SQL-QUERIES BASED ON REGULARIZATION MECHANISM // University News. North-Caucasian Region. Technical Sciences Series. 2019. (4). С. 31–38.

[11] JavaScript // Веб-документация MDN [Электронный ресурс]. URL: https://developer.mozilla.org/ru/docs/Web/JavaScript.

[12] Vue.js // Vuejs.org [Электронный ресурс]. URL: https://vuejs.org/v2/guide/.

[13] NuxtJS [Электронный ресурс]. URL: https://nuxtjs.org/guide#features (дата обращения: 20.05.2020).

[14] Fabio Nelli Beginning JavaScript Charts With JqPlot, D3, and Highcharts / Fabio Nelli, Springer, 2014.

[15] Aendrew Rininsland, Swizec Teller D3.js 4.x data visualization : learn to visualize your data with JavaScript / Aendrew Rininsland, Swizec Teller, Birmingham: Packt, 2017.

[16] Helder Da Rocha Learn D3.js : create interactive data-driven visualizations for the web with the D3.js library / Helder Da Rocha, Birmingham, Uk: Packt Publishing, 2019.

[17] Kuan J. Learning highcharts 4. / J. Kuan, Packt Publishing Limited, 2015.

[18] Vuex // vuex.vuejs.org [Электронный ресурс]. URL: https://vuex.vuejs.org/ru/ (дата обращения: 20.05.2020).

[19] ClickHouse // GitHub [Электронный ресурс]. URL: https://github.com/ClickHouse/ClickHouse/blob/master/docker/test/performance-comparison/performance_comparison.md (дата обращения: 20.05.2020).

[20] Vue CLI // cli.vuejs.org [Электронный ресурс]. URL: https://cli.vuejs.org/guide/ (дата обращения: 20.05.2020).