Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Разработка базы данных для хранения и обработки результатов проведения тестов Тьюринга

Студент ИУ7-65Б (Группа) _____ (Подпись, дата) **С. К. Романов** (И.О.Фамилия)

Руководитель курсового проекта _____ (Подпись, дата) **К.А. Кивва** (И.О.Фамилия)

Консультант _____ (Подпись, дата) **К.А. Кивва** (И.О.Фамилия)

2023 г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие термины с соответствующими определениями.

Natural Language Processing — «Обработка текстов на естественном языке» относится к области компьютерных наук, а точнее к области искусственного интеллекта или ИИ, связанной с предоставлением компьютерам возможность понимать текст и произносимые слова почти так же, как люди. НЛП сочетает в себе вычислительную лингвистику — моделирование человеческого языка на основе правил — со статистическими моделями, машинным обучением и моделями глубокого обучения. Вместе эти технологии позволяют компьютерам обрабатывать человеческий язык в виде текстовых или голосовых данных и «понимать» его полное значение, включая намерения и чувства говорящего или пишущего. [1]

ACID — В контексте обработки транзакций аббревиатура ACID относится к четырем ключевым свойствам транзакции: атомарность (Atomicity), непротиворечивость (Consistency), изоляция (Isolation) и устойчивость (Durability). [2].

Graph Database — база данных, использующая структуры графов для семантических запросов с узлами, ребрами и свойствами для представления и хранения данных. [2].

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие сокращения и обозначения.

NLP — «Natural Language Processing».

ACID — «Atomicity, Consistency, Isolation, Durability».

SQL — «Structured Query Language».

GDB — «Graph Database»

ИИ — «Искусственный Интеллект»

ВВЕДЕНИЕ

Тест Тьюринга — это способ определения способности машины производить интеллектуальные действия, неотличимые от действий человека. В современном мире тесты Тьюринга стали одним из ключевых инструментов для определения искусственного интеллекта. Эти тесты позволяют определить, насколько хорошо компьютер может имитировать разговор человека. Для проведения теста Тьюринга используется программное обеспечение, которое имитирует человеческое поведение и должно убедить эксперта в том, что он общается с живым человеком. Результаты проведения тестов могут быть использованы для разработки и улучшения алгоритмов искусственного интеллекта. Это позволяет разработчикам алгоритмов искусственного интеллекта улучшать свои продукты и технологии.

Примерами таких алгоритмов являются алгоритмы обработки естественных языков (Natural Language Processing — NLP). В общих словах — это совокупность методов и техник, которые позволяют компьютерам анализировать, понимать и генерировать естественный язык. NLP используется в ряде приложений, включая автоматический перевод, распознавание речи и анализ текста. Анализ результатов тестирования поможет в будущем улучшить данные модели, позволяя избегать различные грамматические, орфографические и смысловые ошибки.

Модели от команды OpenAI и многие другие играют важную роль в развитии искусственного интеллекта. GPT-3 неплохо справляется с созданием художественной литературы, поэзии, пресс-релизов, кода, музыки, шуток, технических руководств и новостных статей. Возможно, как предполагает Чалмерс (2020, Other Internet Resources), GPT-3 «предлагает потенциальный бездумный путь к общему искусственному интеллекту». Но, конечно, GPT-3 даже не близок к прохождению теста Тьюринга: на глобальном уровне — учитывая значения нескольких предложений, абзацев или двустороннего диалога — становится очевидным, что GPT-3 не понимает, о чем говорит. У него нет здравого смысла (Common Sense) или способности отслеживать объекты во время обсуждения. Разработанная база данных для хранения и обработки результатов тестов Тьюринга — это хороший инструмент для создания более развитых систем искусственного интеллекта.

Цель данной работы — разработать базу данных для хранения результатов проведения тестов Тьюринга. База данных должна содержать информацию о тестируемых программах, экспертах, результатах проведения тестов и другие данные, необходимые для анализа результатов.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- Определить структуру базы данных и ее таблиц.
- Разработать модели данных для каждой таблицы.
- Написать запросы для вставки, обновления и удаления данных в таблицах.
- Реализовать функционал для получения результатов проведенных тестов, с возможностью фильтрации и сортировки по различным полям.
- Разработать интерфейс пользователя для удобного использования базы данных.

1 Аналитический раздел

В данном разделе описана структура теста Тьюринга. Представлен анализ способов хранения данных и систем управления базами данных, оптимальных для решения поставленной задачи.

1.1 Формализация задачи

Тест Тьюринга — это метод оценки способности машины производить интеллектуальные действия, сравнивая ее поведение с поведением человека в решении задач. Тест заключается в том, что человек задает вопросы другому человеку и компьютеру, а затем пытается определить, от кого пришли ответы. Если компьютер может убедительно имитировать поведение человека, то он считается способным производить интеллектуальные действия.

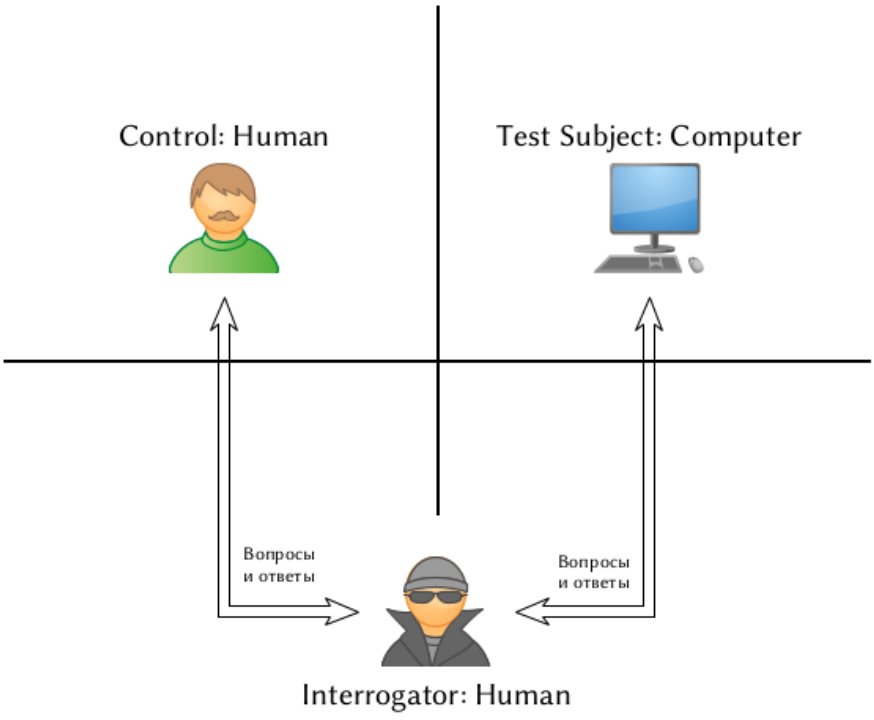

Рис. 1: Тест Тьюринга

Тьюринг в своей работе [3] описывает следующий вид игры. Предположим, что у нас есть человек, машина и эксперт. Эксперт находится в комнате, отделенной от другого человека и машины. Цель игры состоит в том, чтобы

эксперт определил, кто из двух является человеком, а кто машиной. Эксперт знает человека и машину по меткам «X» и «Y» — но, по крайней мере в начале игры, не знает, кто из них человек и кто – машина — и в конце игры он должен сказать либо «X — это человек, а Y — машина», либо «X — это машина, а Y — человек». Эксперту разрешается задавать человеку и машине вопросы следующего вида: «Скажите, пожалуйста, X, играет ли X в шахматы?» Кто бы из машины и другого человека ни был X, он должен отвечать на вопросы, адресованные X. Цель машины состоит в том, чтобы попытаться заставить эксперта ошибочно заключить, что машина — это другой человек; цель другого человека состоит в том, чтобы попытаться помочь эксперту правильно идентифицировать машину. [4]

Следует отметить, что во времена Тьюринга, было ограничение, что ответы поступали через ограниченные врменные рамки, поскольку время ответа компьютера было гораздо больше, чем у человека. Сегодня это ограничение сохраняется, однако из-за обратного: реакция компьютера быстрее, чем реакция человека.

1.2 Способы хранения данных

Для решения задачи хранения теста Тьюринга необходимо хранить следующие данные:

1. Данные о человеке, машине и эксперте;
2. Данные о заданных вопросах и полученных ответах;
3. Данные, о связях между вопросами и ответами.

Поскольку в конце теста выносится вердикт о том, является ли отвечающий машиной или человеком, необходимо также хранить какие ответы были даны в каком порядке и каким актором. Эти данные должны быть доступны для обработки и сравнения в процессе игры.

Один из способов хранения данных — это использование реляционных баз данных. В этом случае можно создать таблицы для каждого объекта (люди, машины и эксперты, вопросы и ответы, и т.д.) и связать их отношениями.

Например, таблицы «Person», «Computer» и «Interrogator» могут быть связаны через внешние ключи. Это является надежным и проверенным способом хранения данных.

Однако более эффективный способ хранения данных — это использование графовых баз данных (GDB), таких как SurrealDB или Neo4j. В этом случае каждый объект может быть представлен узлом графа, а отношения между объектами — ребрами графа 2.

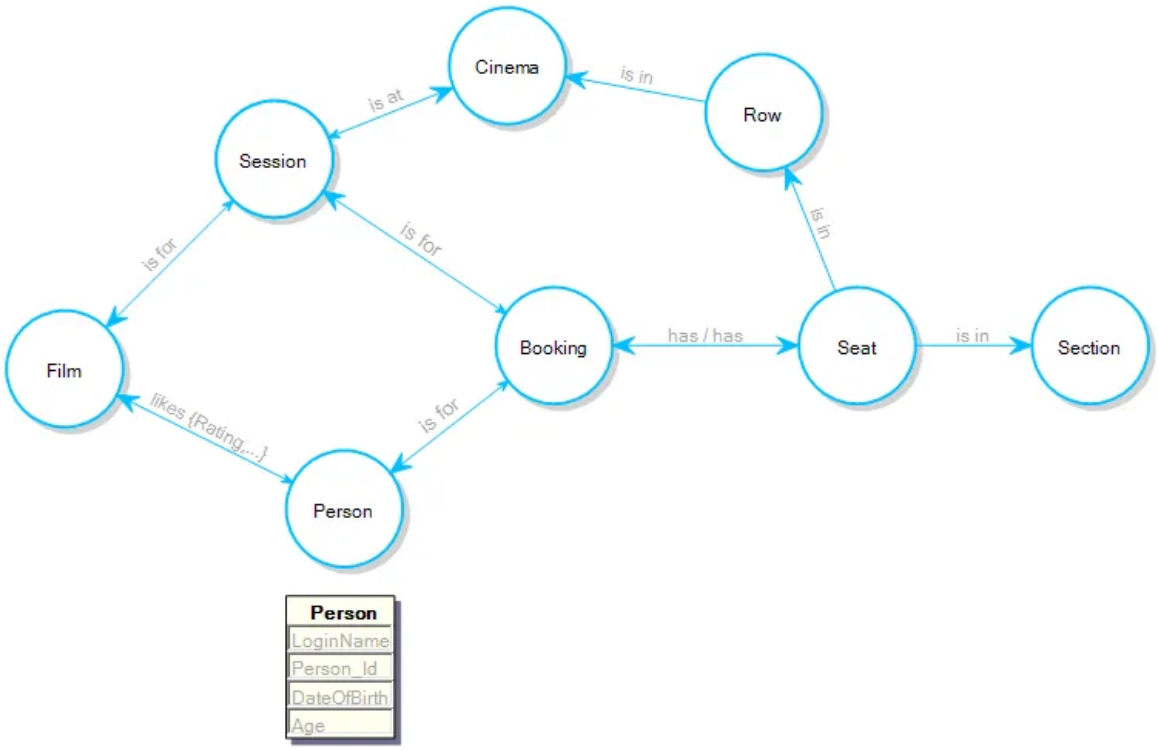

Рис. 2: Представление базы данных в виде графа

Ключевым понятием системы является граф (или ребро, или взаимосвязь). Граф связывает элементы данных в хранилище с набором узлов и ребер, причем ребра представляют отношения между узлами. Отношения позволяют напрямую связывать данные в хранилище и во многих случаях извлекать их с помощью одной операции. Базы данных графов удерживают отношения между данными в качестве приоритета. Запрашивать отношения быстро, потому что они постоянно хранятся в базе данных. Отношения можно интуитивно визуализировать с помощью графов, что делает их полезными для сильно взаимосвязанных данных [5]. Поскольку графовая модель данных более естественным образом отображает связи между объектами, это делает ее более подходящей для задач, связанных с анализом связей и отношений между данными. В графовых базах данных нет необходимости использовать сложные

JOIN-запросы, что может существенно упростить запросы к данным.

Графовые базы данных также обеспечивают быстрый доступ к данным по отношениям, что делает их эффективными при работе с глубоко связанными данными. Они также позволяют легко добавлять новые данные в граф без необходимости изменения схемы базы данных. Однако, реляционные базы данных обладают более высокой надежностью и могут обеспечивать лучшую производительность при выполнении сложных запросов, особенно если используются правильно настроенные индексы.

Таким образом, выбор между реляционными базами данных и графовыми зависит от конкретных требований проекта. В контексте данной работы, графовая модель подходит больше, чем реляционная, потому что тест Тьюринга включает в себя множество связей между объектами (человек, машина, эксперт, вопросы и ответы и т.д.), которые можно представить в виде графа.

Графовая модель также становится еще более привлекательной, если вспомнить какая идея была обозначена в начале данной работы: создание инструмента для улучшения искусственного интеллекта. В большинстве случаев ИИ работает не со стандартными «табличными» данными, а данными, представленных в виде графа. Таким образом, схожая структура данных внутри СУБД поможет разработать более гибкую и быстродействующую систему при меньших затраченных ресурсах

На рис. 3 можно увидеть результаты сравнения 3 различных баз данных: реляционной (PostgreSQL), графовой (Neo4j) и мультимодельной (ArangoDB).

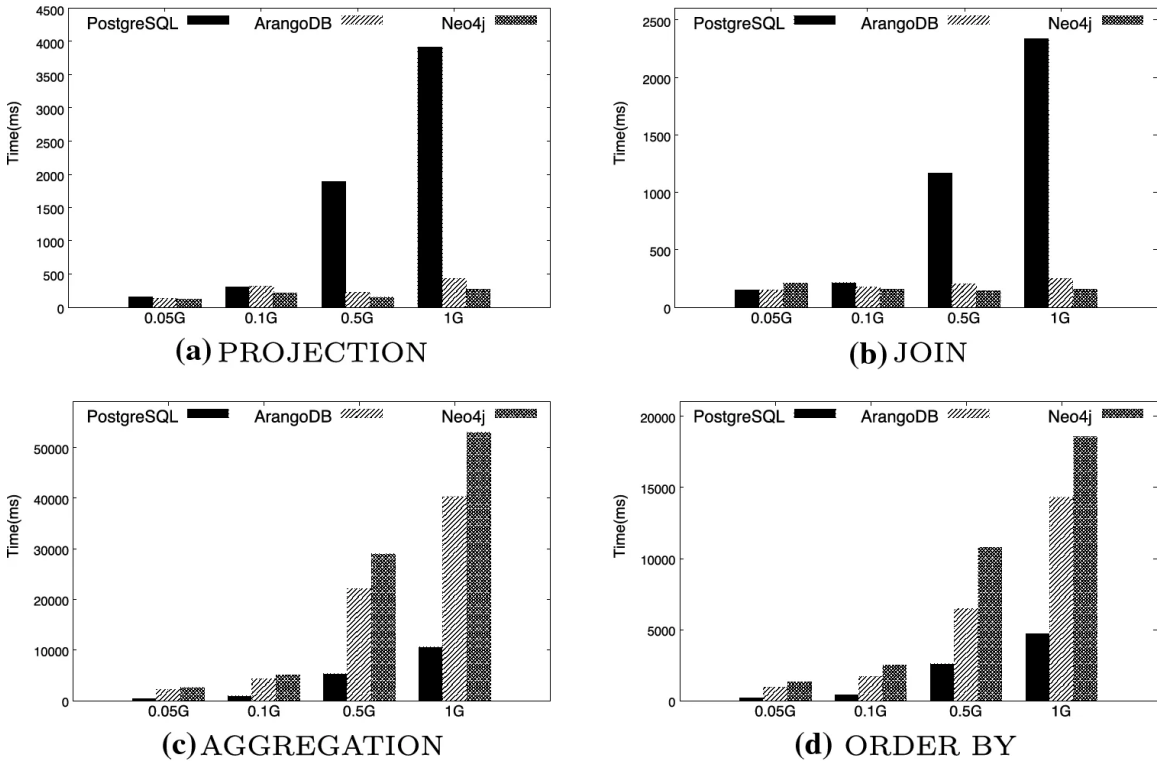

Рис. 3: Сравнение времени работы различных баз данных над атомарными операциями.

1.3 Системы управления базами данных

Для выбора системы управления базами данных необходимо учитывать требования к производительности и масштабируемости приложения.

Реляционные базы данных имеют высокую надежность и поддерживают ACID-свойства транзакций. Они также обеспечивают хорошую производительность при выполнении сложных запросов. Однако, они требуют дополнительного управления индексами и ключевыми полями.

Графовые базы данных обеспечивают высокую производительность при работе с глубоко связанными данными. Граф связывает элементы данных в хранилище с набором узлов и ребер, причем сами ребра представляют отношения между узлами. Отношения позволяют напрямую связывать данные в хранилище и во многих случаях извлекать их с помощью одной операции. Отношения между данными в подобных системах имеют приоритет над самими данными, поэтому запрос по отношениям является крайне быстрой операцией, поскольку они постоянно хранятся в базе данных. Также отношения можно интуитивно визуализировать с помощью графов, что делает их полезными для сильно взаимосвязанных данных.

1.3.1 SurrealDB

SurrealDB — мультимодельная NewSQL база данных, которая работает в режиме полной схемы (`SCHEMAFULL`) или без схемы (`SCHEMALESS`), с таблицами, ссылками на записи между документами (без `JOIN`) и функциями моделирования базы данных на основе графов [6].

Благодаря использованию SurrealDB особых методов сегментирования и репликации, становится возможным повысить производительность за счет распределения нагрузки между несколькими компьютерами [7].

Также особая архитектура базы данных позволяет работать как в оперативной памяти (`in-memory`), на дисковом пространстве (`on-disk`) или как распределенная база данных, используя TiKV [8].

Поскольку SurrealDB — мультимодельная база данных, становится также возможным классические реляционные методики проектирования баз данных, что повышает гибкость итоговой системы.

1.3.2 Neo4j

Neo4j — это графовая база данных, которая позволяет хранить, управлять и анализировать связанные данные. Она была разработана с учетом графовой модели данных, в которой данные представлены в виде узлов (вершин) и связей (ребер) [9].

Одним из преимуществ Neo4j является то, что она позволяет эффективно моделировать и анализировать сложные связи между данными, такие как социальные сети, географические карты и сети предприятий. Это делает ее очень полезной для приложений, которые требуют быстрого доступа к сложным связным данным и быстрой обработки запросов.

Однако поскольку Neo4j — исключительно графовая база данных, хранение и получение данных без каких-либо связей друг с другом может вызвать проблемы с производительностью, вне зависимости от размера запроса [10].

1.4 Выбор СУБД для решения задачи

Для решения задачи теста Тьюринга необходимо выбрать графовую базу данных, поскольку графовая модель данных более естественным образом отображает связи между объектами.

Среди графовых баз данных можно выделить две наиболее подходящие системы: SurrealDB и Neo4j. Обе СУБД обеспечивают быстрый доступ к данным по отношениям, что делает их эффективными при работе с глубоко связанными данными.

Однако SurrealDB имеет дополнительные преимущества перед Neo4j. Она является мультимодельной базой данных, что позволяет эффективное хранение и получение несвязанных данных, где Neo4j может испытывать определенные проблемы.

Вывод

В данном разделе:

– рассмотрена сущность и стурктура теста Тьюринга;

– проанализированы способы хранения информации для системы и выбраны оптимальные способы для решения поставленной задачи;

– были рассмотрены два различных типа баз данных: реляционные и графовые;

– было выявлено, что для решения задачи теста Тьюринга наиболее подходящей является графовая база данных, а конкретно SurrealDB.

2 Конструкторский раздел

В данном разделе представлены этапы проектирования выделенных в предыдущем разделе баз данных, нужных для решения задачи

2.1 Проектирование базы данных для хранения Тестов Тьюринга

База данных для хранения Тестов Тьюринга будет реализована с использованием СУБД SurrealDB. В базе данных будет существовать 7 сущностей и 7 типов отношений. ER-диаграма сущностей этой базы данных представлена на рисунке.

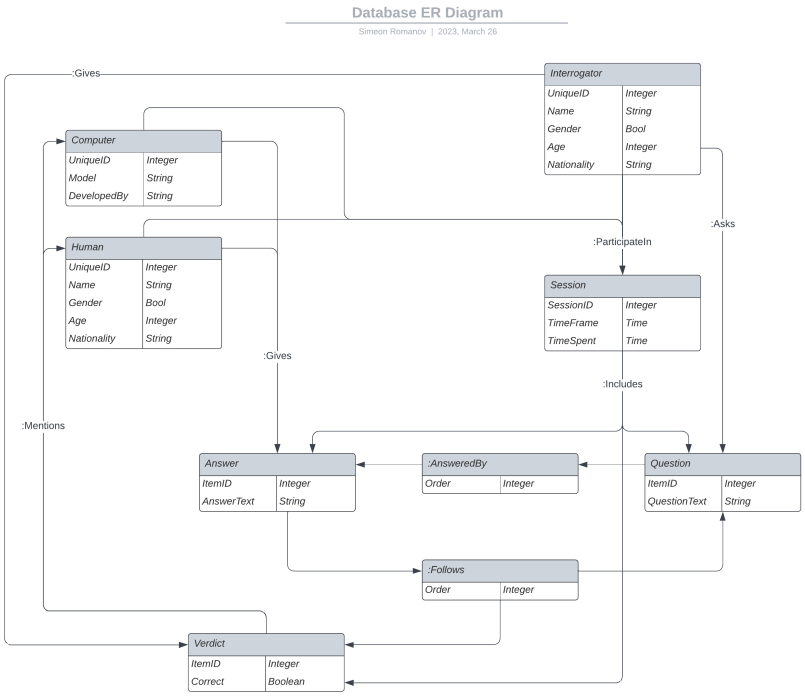

Рис. 4: Диаграмма сущностей и отношений базы данных.

Поля вершины `Interrogator`:

1. `UniqueID` — уникальный идентификатор, присваемый сущности-субъекту; будет использоваться чтобы однозначно идентифицировать сущности, способные принимать решения в системе;

2. Name — имя «эксперта»;

3. Gender — пол «эксперта»;

4. Age — возраст «эксперта»;

5. Nationality — национальность «эксперта».

Данная таблица отвечает за хранение данных, связанных с экспертом, проводящим эксперимент. Эта таблица связана со следующими таблицами:

– Session — через отношение :ParticipateIn;

– Answer — через отношение :Asks;

– Verdict — через отношение :Gives.

Поля вершины Computer:

1. UniqueID — уникальный идентификатор, присваемый сущности-субъекту; будет использоваться чтобы однозначно идентифицировать сущности, способные принимать решения в системе;

2. Model — Модель ИИ, проходившая тест;

3. DevelopedBy — Разработчики указанной ИИ.

Данная таблица отвечает за хранение данных, связанных с компьютером, участвующем в эксперименте. Эта таблица связана со следующими таблицами:

– Session — через отношение :ParticipateIn;

– Answer — через отношение :Gives;

– Verdict. — через отношение :Mentions.

Поля вершины Human:

1. UniqueID — уникальный идентификатор, присваемый сущности-субъекту; будет использоваться чтобы однозначно идентифицировать сущности, способные принимать решения в системе;

2. Name — имя человека;

3. Gender — пол человека;

4. Age — возраст человека;

5. Nationality — национальность человека.

Данная таблица отвечает за хранение данных, связанных с человеком, участвующем в эксперименте. Эта таблица связана со следующими таблицами:

– Session — через отношение :ParticipateIn;

– Answer — через отношение :Gives;

– Verdict — через отношение :Mentions.

Поля вершины Answer:

1. ItemID — уникальный идентификатор, присваемый сущности-объекту; будет использоваться чтобы однозначно идентифицировать сущности, которые являются производными от объектов.

2. AnswerText — текст ответа.

Данная таблица отвечает за хранение данных, связанных с ответами, данными в эксперименте. Следует отметить, что ответы, данные на протяжении всех экспериментов, являются уникальными сущностями, или иным словами, в базе данных нет двух одинаковых ответов на любой из вопросов. Данная особенность преследует цель показать связь между вопросами и ответами и как различные вопросы могут привести к одним и тем же ответам, либо же, как компьютер и человек в экперименте могут дать одинаковый ответ.

Эта таблица связана со следующими таблицами:

– Session — через отношение :Includes;

– Answer — через отношения :AnsweredBy и :Follows;

– Computer — через отношение :Gives;

– Human — через отношение :Gives;

– Verdict — через отношение :Follows.

Поля вершины Question:

1. ItemID — уникальный идентификатор, присваемый сущности-объекту; будет использоваться чтобы однозначно идентифицировать сущности, которые являются производными от объектов.

2. QuestionText — текст вопроса.

Данная таблица отвечает за хранение данных, связанных с вопросами, данными в эксперименте экспертом. Следует отметить, что вопросы, данные на протяжении всех экспериментов, как и ответы, упомянутые выше, являются уникальными сущностями, или иным словами, в базе данных нет двух одинаковых вопросов. Данная особенность преследует цель показать связь между вопросами и ответами и как на один вопрос можно привести множество различных ответов.

Эта таблица связана со следующими таблицами:

– Session — через отношение :Includes;

– Answer — через отношения :AnsweredBy и :Follows;

– Interrogator — через отношение :Asks.

Поля вершины Verdict:

1. ItemID — уникальный идентификатор, присваемый сущности-объекту; будет использоваться чтобы однозначно идентифицировать сущности, которые являются производными от объектов;

2. Correct — Верен ли вердикт, выданный экспертом.

Данная таблица отвечает за хранение данных, связанных с вердиктами, данными экспертами по окончанию экспериментов. После любого данного ответа, эксперт может закончить эксперимент и выдать свой вердикт, кто является компьютером, а кто человеком.

Эта таблица связана со следующими таблицами:

– Session — через отношение :Includes;

– Answer — через отношение :Follows;

– Interrogator — через отношение :Gives;

– Computer — через отношение :Mentions;

– Human — через отношение :Mentions.

Поля вершины Session:

1. SessionID — уникальный идентификатор, присваемый сессии;

2. TimeFrame — период времени, отведенный на ответ на вопрос;

3. TimeSpent — продолжительность сессии.

Данная таблица отвечает за хранение данных, связанных с различными экспериментами. Данная таблица является свеого рода meta-таблицей, по связи с которой можно получить данные о всех сущностях, участвующими в эксперименте.

Эта таблица связана со следующими таблицами:

– Question — через отношение :Includes;

– Answer — через отношение :Includes;

– Answer — через отношение :Includes;

– Interrogator — через отношение :ParticipateIn;

– Computer — через отношение :ParticipateIn;

– Human — через отношение :ParticipateIn.

Поля рёбер :AnsweredBy и :Follows:

1. Order — порядковый номер вопроса/ответа/вердикта в системе

Особенность SurrealDB заключается в том, что отношения также могут иметь дополнительные поля, характеризующие их. Поле Order необходимо для построения контекста ответов/вопросов поскольку ответ может различаться от того, какие ответы были даны ранее.

Вывод

В данном разделе были представлены этапы проектирования баз данных и рассмотрены особенности используемой СУБД на архитектурном уровне.

3 Технологический раздел

В данном разделе представлены

– архитектура;

– средства разработки программного обеспечения;

– детали реализации;

– способы взаимодействия с программным продуктом.

3.1 Архитектура

Предполагается, что разрабатываемый проект является одним цельным Electron-подобным [11] приложением. Серверная часть и графический интерфейс упаковывается в единый бинарный файл, предоставляя возможность создать различные версии приложения под различные операционные системы. В общем смысле, серверная часть коммуницирует с базой данных, доставляя результат к графическому интерфейсу в рамках единого приложения.

Общая схема архитектура приложения представлена на рисунке 5

3.2 Средства реализации

Основным языком программирования является мультипарадигменный язык Rust [12].

– Одно из главных достоинств данного языка это гарантия безопасной работы с памятью при помощи системы статической проверки ссылок, так называемый Borrow Checker [13].

– Отсутствие сборщика мусора, как следствие, более экономная работа с ресурсами.

– Встроенный компилятор, поставляемый совместно с пакетным менеджером Cargo.

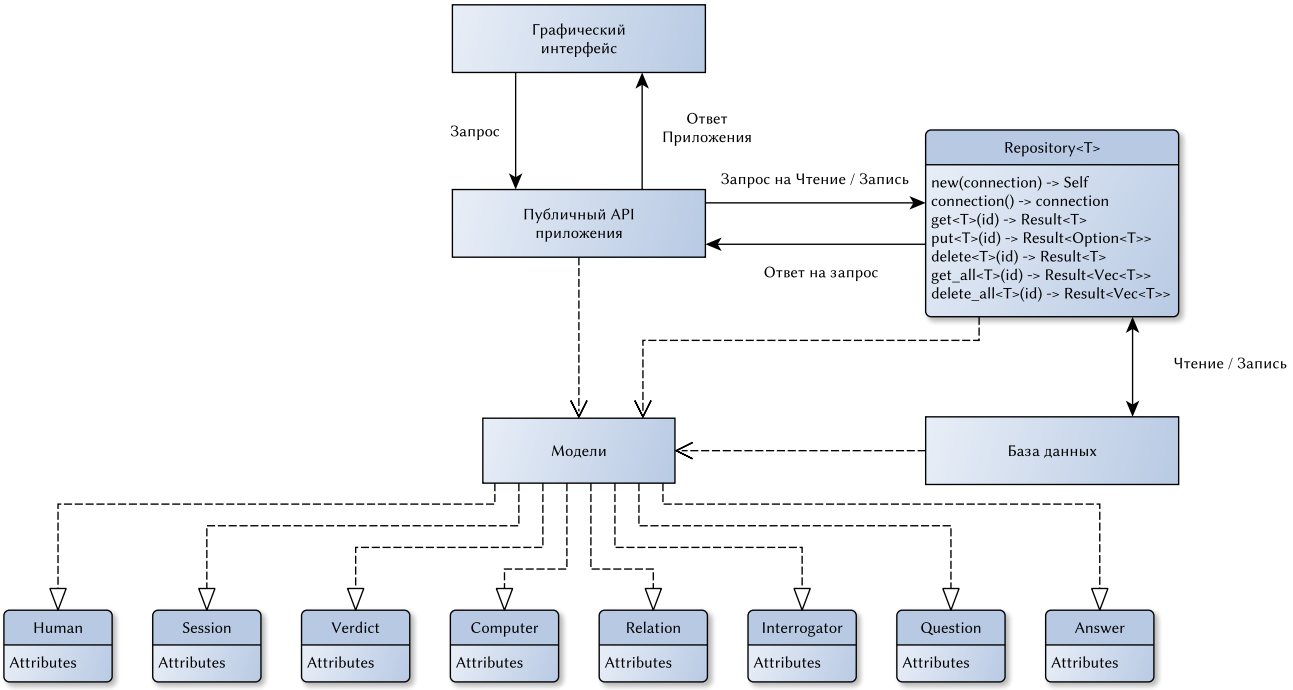

Рис. 5: Схема архитектуры приложения

– Кросс-платформенность, от UNIX и MacOS приложений до Web - приложений.

– `SurrealDB` и `Rust` написаны на одном и том же языке, в следствии чего инструментарий наиболее плотно работает с непосредственно самой базой данных.

– Важно отметить, что язык программирования Rust сопоставим по скорости с такими языками как C и C++, предоставляя в то же время более широкий функционал для тестирования кода и контроля памяти.

Также в рамках языка `Rust` был выбран фреймворк `Tauri`. `Tauri` используется для создания приложений с использованием комбинации инструментов `Rust` и `HTML`, отображаемых в `Webview`. Приложения, созданные с помощью `Tauri`, могут поставляться с любым количеством дополнительных `JS API` и `Rust API`, так что `Webview` может управлять системой посредством передачи сообщений. Разработчики могут расширить API за счет своей собственной функциональности и легко объединить `Webview` и серверную часть на основе `Rust` [14]. На рисунке 6 изображена общая архитектура `Tauri`-приложения.

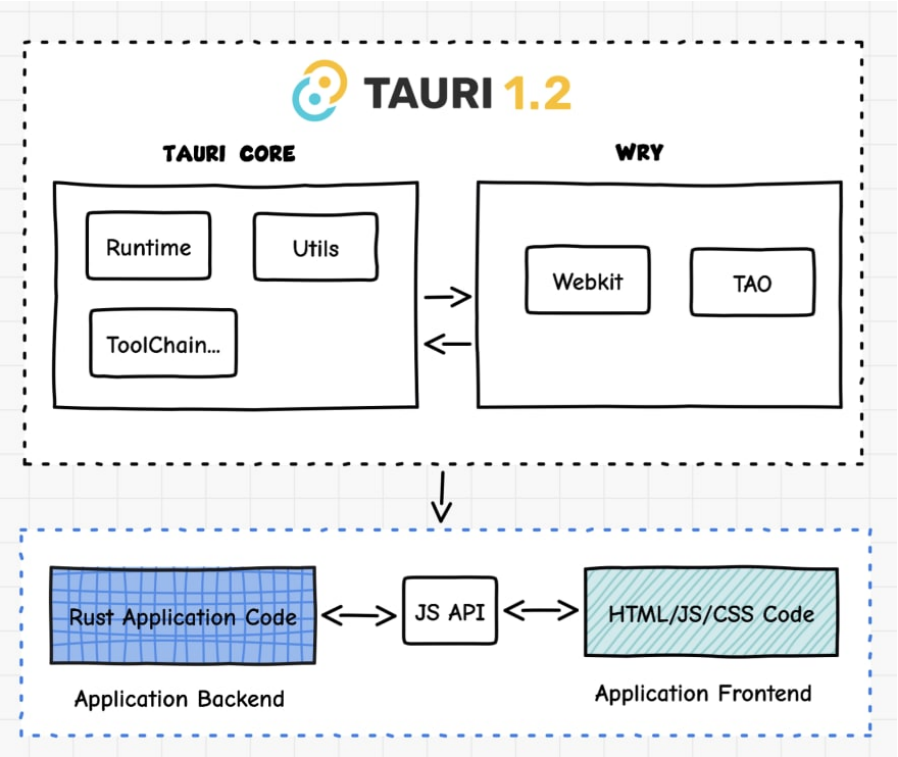

Рис. 6: Принцип работы Tauri

3.3 Детали реализации

На листинге 1 представлены трейты, аналог интерфейса в языке Rust, необходимые для реализации репозитория, который отвечает за взаимодействие между публичным API и непосредственно с базой данных.

Листинг 1: Трейты, необходимые для реализации репозитория.

```rust
use crate::models::{HasRole, User};
use crate::prelude::*;
use ::surrealdb::opt::auth::Jwt;
use std::error::Error;
use uuid::Uuid;

pub mod surrealdb;

pub trait CrudOps<T> {
    async fn get(&self, id: Uuid) -> Result<T, Box<dyn Error>>;
    async fn put(&self, id: Uuid, value: T) -> Result<Option<T>, Box<dyn Error>>;
    async fn delete(&self, id: Uuid) -> Result<T, Box<dyn Error>>;
    async fn get_all(&self) -> Result<Vec<T>, Box<dyn Error>>;
    async fn delete_all(&self) -> Result<Vec<T>, Box<dyn Error>>;
}

pub trait MetaOps {
```

```
    async fn get_meta(&self) -> Result<User, Box<dyn Error>>;
}

pub trait Repository<T, C>
where
    C: CrudOps<T>,
{
    fn new(connection: C) -> Result<Self>
    where
        Self: std::marker::Sized;
    fn connection(&self) -> C;
    async fn get<G>(&self, id: Uuid) -> Result<T, Box<dyn Error>> {
        self.connection().get(id).await
    }
    async fn put(&self, id: Uuid, value: T) -> Result<Option<T>, Box<dyn Error>> {
        self.connection().put(id, value).await
    }

    async fn delete(&self, id: Uuid) -> Result<T, Box<dyn Error>> {
        self.connection().delete(id).await
    }

    async fn get_all(&self) -> Result<Vec<T>, Box<dyn Error>> {
        self.connection().get_all().await
    }

    async fn delete_all(&self) -> Result<Vec<T>, Box<dyn Error>> {
        self.connection().delete_all().await
    }
}

pub trait Utils<T, C>
where
    Self: Repository<T, C>,
    C: MetaOps + CrudOps<T>,
    T: HasRole,
{
    async fn meta_from_jwt(&self, token: Jwt) -> Result<User, Box<dyn Error>> {
        self.connection().get_meta().await
    }
}
```

На листинге 2 представлена реализация репозитория и моделей, необходимых для трансляции данных из SurrealDB в пространство языка Rust.

Листинг 2: Реализация репозитория.

```
impl<T: DeserializeOwned + Serialize + Send + Sync + HasId> CrudOps<T> for
    Surreal<Client> {
```

```
    async fn get(&self, id: Uuid) -> Result<T, Box<dyn Error>> {
        let result: Option<T> = self
            .select((
                std::any::type_name::<T>()
                    .to_lowercase()
                    .rsplit("::")
                    .next()
                    .unwrap_or(&std::any::type_name::<T>().to_lowercase()),
                id.to_string(),
            ))
            .await?;

        result.map_or_else(
            || {
                tracing::warn!("GET: field not found");
                Err(Err::GetNotFound {
                    table: std::any::type_name::<T>()
                        .to_lowercase()
                        .rsplit("::")
                        .next()
                        .unwrap_or(&std::any::type_name::<T>().to_lowercase())
                        .to_string(),
                    id,
                }
                .into())
            },
            |mut res| {
                tracing::info!("GET: success");
                *res.id() = id;
                Ok(res)
            },
        )
    }

    async fn put(&self, id: Uuid, value: T) -> Result<Option<T>, Box<dyn Error>> {
        let result: Option<T> = self
            .update((
                std::any::type_name::<T>()
                    .to_lowercase()
                    .rsplit("::")
                    .next()
                    .unwrap_or(&std::any::type_name::<T>().to_lowercase()),
                id.to_string(),
            ))
            .content(value)
            .await?;

        Ok(result)
```

```
    }

    async fn delete(&self, id: Uuid) -> Result<T, Box<dyn Error>> {
        let result: Option<T> = self
            .delete((
                std::any::type_name::<T>()
                    .to_lowercase()
                    .rsplit("::")
                    .next()
                    .unwrap_or(&std::any::type_name::<T>().to_lowercase()),
                id.to_string(),
            ))
            .await?;

        result.map_or_else(
            || {
                tracing::warn!("DELETE: field not found");
                Err(Err::GetNotFound {
                    table: std::any::type_name::<T>()
                        .to_lowercase()
                        .rsplit("::")
                        .next()
                        .unwrap_or(&std::any::type_name::<T>().to_lowercase())
                        .to_string(),
                    id,
                }
                .into())
            },
            |res| {
                tracing::info!("DELETE: success");
                Ok(res)
            },
        )
    }

    async fn get_all(&self) -> Result<Vec<T>, Box<dyn Error>> {
        Ok(self
            .select(
                std::any::type_name::<T>()
                    .to_lowercase()
                    .rsplit("::")
                    .next()
                    .unwrap_or(&std::any::type_name::<T>().to_lowercase()),
            )
            .await?)
    }

    async fn delete_all(&self) -> Result<Vec<T>, Box<dyn Error>> {
```

```
        Ok(self
            .delete(
                std::any::type_name::<T>()
                    .to_lowercase()
                    .rsplit("::")
                    .next()
                    .unwrap_or(&std::any::type_name::<T>().to_lowercase()),
            )
            .await?)
    }
}

// impl<T: DeserializeOwned> MetaOps<T> for Surreal<Client> {
impl MetaOps for Surreal<Client> {
    async fn get_meta(&self) -> Result<User, Box<dyn Error>> {
        let res: SurrealUser = self
            .select(("user"))
            .await?
            .pop()
            .ok_or(Box::from("No meta found") as Box<dyn Error>)?;

        res.try_into()
    }
}

default impl<T: DeserializeOwned + Serialize + Send + Sync + HasId> Repository<T,
    Surreal<Client>>
    for SurrealRepo<T>
{
    fn new(connection: Surreal<Client>) -> Result<Self> {
        Ok(Self {
            connection,
            object: PhantomData::<T>,
        })
    }

    fn connection(&self) -> Surreal<Client> {
        self.connection.clone()
    }
}

// Needed for specialization
impl Repository<Human, Surreal<Client>> for SurrealRepo<Human> {}
impl Repository<Interrogator, Surreal<Client>> for SurrealRepo<Interrogator> {}
impl Repository<Computer, Surreal<Client>> for SurrealRepo<Computer> {}
impl Repository<Answer, Surreal<Client>> for SurrealRepo<Answer> {}
impl Repository<Question, Surreal<Client>> for SurrealRepo<Question> {}
impl Repository<Session, Surreal<Client>> for SurrealRepo<Session> {}
```

```
145 impl Repository<Verdict, Surreal<Client>> for SurrealRepo<Verdict> {}
```

На листинге 3 представлены запросы, которые вызываются при инициализации базы данных.

Листинг 3: Инициализация базы данных.

```
1  REMOVE DATABASE TuringDB;
2  REMOVE NAMESPACE TuringApp;
3
4  DEFINE NAMESPACE TuringApp;
5  USE NS TuringApp;
6  DEFINE DATABASE TuringDB;
7  USE DB TuringDB;
8
9  DEFINE TABLE role SCHEMAFULL
10     PERMISSIONS
11       FOR create, update NONE,
12       FOR select WHERE $auth.roles containsany [role:human, role:interrogator,
       role:computer],
13       FOR delete NONE;
14   create role:human;
15   create role:computer;
16   create role:interrogator;
17
18 DEFINE TABLE user SCHEMAFULL
19     PERMISSIONS
20         FOR select, update WHERE id = $auth.id,
21         FOR create, delete NONE;
22   DEFINE FIELD user ON user TYPE string;
23   DEFINE FIELD password ON user TYPE string;
24   DEFINE FIELD role ON user TYPE record;
25   DEFINE INDEX idx_user ON user COLUMNS user UNIQUE;
26
27 DEFINE SCOPE TuringScope
28     SESSION 1h
29     SIGNUP ( CREATE user SET id = rand::uuid(), user = $user, password =
       crypto::argon2::generate($password), role = $role )
30     SIGNIN ( SELECT * FROM user WHERE user = $user AND crypto::argon2::compare(password,
       $password) );
31
32 DEFINE TABLE human SCHEMAFULL
33     PERMISSIONS
34         FOR select WHERE true,
35         FOR create, delete, update WHERE id = $auth.id AND $auth.role containsany
       [role:human];
36   DEFINE FIELD name ON human TYPE string;
37   DEFINE FIELD age ON human TYPE int;
38   DEFINE FIELD gender ON human TYPE string;
```

```
  DEFINE FIELD nationality ON human TYPE string;

DEFINE TABLE interrogator SCHEMAFULL
    PERMISSIONS
        FOR select WHERE true,
        FOR create, delete, update WHERE id = $auth.id AND $auth.role containsany
    [role:interrogator];
  DEFINE FIELD name ON interrogator TYPE string;
  DEFINE FIELD age ON interrogator TYPE int;
  DEFINE FIELD gender ON interrogator TYPE string;
  DEFINE FIELD nationality ON interrogator TYPE string;

DEFINE TABLE verdict SCHEMAFULL
    PERMISSIONS
        FOR select WHERE true,
        FOR create, delete, update WHERE ->gives->id = $auth.id AND $auth.role
    containsany [role:interrogator];
  DEFINE FIELD correct ON verdict TYPE bool;

DEFINE TABLE session SCHEMAFULL
    PERMISSIONS
        FOR select WHERE true,
        FOR create, delete, update WHERE ->participateIn->id = $auth.id AND $auth.role
    containsany [role:interrogator];
  DEFINE FIELD time_start ON session TYPE string DEFAULT time::now();
  DEFINE FIELD time_end ON session TYPE string DEFAULT time::now();
  DEFINE FIELD time_spent ON session TYPE string DEFAULT 0;

DEFINE TABLE question SCHEMAFULL
    PERMISSIONS
        FOR select WHERE true,
        FOR create, delete, update WHERE ->asks->id = $auth.id AND $auth.role
    containsany [role:interrogator];
  DEFINE FIELD text ON question TYPE string;

DEFINE TABLE computer SCHEMAFULL
    PERMISSIONS
        FOR select WHERE true,
        FOR create, delete, update WHERE id = $auth.id AND $auth.role containsany
    [role:computer];
  DEFINE FIELD model ON computer TYPE string;
  DEFINE FIELD developed_by ON computer TYPE string;

DEFINE TABLE answer SCHEMAFULL
    PERMISSIONS
        FOR select WHERE true,
        FOR create, delete, update WHERE ->gives->id = $auth.id
                                        AND $auth.role containsany [role:computer,
```

```
    role:human];
  DEFINE FIELD text ON answer TYPE string;

DEFINE TABLE mentions;
DEFINE TABLE asks;
DEFINE TABLE gives;
DEFINE TABLE includes;
DEFINE TABLE participateIn;
DEFINE TABLE follows;
DEFINE TABLE answeredBy;

DEFINE FIELD order ON follows TYPE int;
DEFINE FIELD answeredBy ON follows TYPE int;

DEFINE INDEX mentions ON TABLE mentions COLUMNS in, out;
DEFINE INDEX asks ON TABLE asks COLUMNS in, out;
DEFINE INDEX gives ON TABLE gives COLUMNS in, out;
DEFINE INDEX includes ON TABLE includes COLUMNS in, out;
DEFINE INDEX participateIn ON TABLE participateIn COLUMNS in, out;
DEFINE INDEX follows ON TABLE follows COLUMNS in, out;
DEFINE INDEX answeredBy ON TABLE answeredBy COLUMNS in, out;
```

На листинге 4 представлены 3 метода API, которые вызываются со стороны графического интерфейса.

Листинг 4: Методы API.

```rust
#[derive(Deserialize)]
pub struct Context {
    pub host: String,
    pub port: u16,
    pub ns: String,
    pub db: String,
    pub sc: String,
}

#[tauri::command]
pub async fn login(
    Credentials {
        username,
        password,
        host,
        port,
        ns,
        db,
        sc,
    }: Credentials,
) -> Result<Jwt, Err> {
    let connection = Surreal::new::<Ws>(format!("{}:{}", host, port)).await?;
```

```
    let res = connection
        .signin(Scope {
            namespace: &ns,
            database: &db,
            scope: &sc,
            params: LoginParams {
                user: &username,
                password: &password,
            },
        })
        .await?;

    Ok(res)
}

#[tauri::command]
pub async fn signup(
    Credentials {
        username,
        password,
        host,
        port,
        ns,
        db,
        sc,
    }: Credentials,
    role: Role,
) -> Result<Jwt, Err> {
    tracing::info!("received Credentials: {ns}, {db}, {username}, {host}, {port},
    {role:?}");
    let connection = Surreal::new::<Ws>(format!("{}:{}", host, port)).await?;
    connection.use_ns(&ns).use_db(&db).await?;
    let sc = Scope {
        namespace: &ns,
        database: &db,
        scope: &sc,
        params: AuthParams {
            user: &username,
            password: &password,
            role: role.into(),
        },
    };

    Ok(connection.signup(sc).await?)
}

#[tauri::command]
pub async fn get_info(
```

```
    Context {
        host,
        port,
        ns,
        db,
        sc,
    }: Context,
    token: Jwt,
) -> Result<User, Err> {
    tracing::info!("received Credentials: {ns}, {db}, {host}, {port}");
    let connection = Surreal::new::<Ws>(format!("{}:{}", host, port)).await?;
    connection.use_ns(&ns).use_db(&db).await?;
    connection.authenticate(token).await?;

    connection.get_meta().await.map_err(Err::General)
}
```

Вывод

В данном разделе были представлена архитектура и средства реализации программного обеспечения, листинги ключевых компонентов системы и пример работы приложения.

4 Исследовательский раздел

В данном разделе приведены описание эксперимента и технические характеристики устройства, на котором проводилось измерение времени работы программного обеспечения, а также результаты замеров времени.

4.1 Постановка эксперимента

В данном подразделе представлены цель, описание и результаты эксперимента.

4.1.1 Цель эксперимента

Целью эксперимента является сравнение времени, требуемого для получения сильносвязанных данных о тесте Тьюринга в двух базах данных `SurrealDB` и `PostgresQL`.

4.1.2 Описание эксперимента

Сравнить занимаемое время для получения данных для различных баз данных можно при помощи бенчмарков — специальных функций, котороые проводят серии различных испытыний с записью производительности системы для дальнейшего их сравнения. Для `SurrealDB` в рамках бенчмаркинга были написаны запросы, утилизирующие её графовую составляющую. В то же время запросы к `PostgresQL` применяют множественные JOIN-запросы ввиду сильносвязанности данных.

Для замера производительности двух ралзичных баз данных при выполнении запросов будет использоваться библиотека `Criterion`, функции которой использовались для определения эффективности запросов по времени.

4.1.3 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

– Операционная система: Arch Linux [15] 64-bit;

– Количество ядре: 4 физических и 8 логических ядер;

– Оперативная память: 16 Гб, DDR4;

– Процессор: 11th Gen Intel® Core™ i5-11320H @ 3.20 ГГц [16].

Во время тестирования устройство было нагружено только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.1.4 Результаты эксперимента

В таблицах 1 - 3 представлены результаты поставленного эксперимента, где сравнивается время исполнения в зависимости от количесвта сущностей в базе данных.

Таблица 1: Результаты сравнения времени, для запросов к `PostgresQL` и `SurrealDB` (количество элементов - 100 единиц)

Количество запросов	**`SurrealDB`, мс**	**`PostgresQL`, мс**
1	58732	45812
5	255079	224391
10	527629	473282
25	1428489	1262168
100	5021948	4624102

Таблица 2: Результаты сравнения времени, для запросов к PostgresQL и SurrealDB (количество элементов - 1000 единиц)

Количество запросов	**SurrealDB, мс**	**PostgresQL, мс**
1	63418	68719
5	284249	301548
10	577324	598925
25	1531132	1762836
100	6145253	7843628

Таблица 3: Результаты сравнения времени, для запросов к PostgresQL и SurrealDB (количество элементов - 5000 единиц)

Количество запросов	**SurrealDB, мс**	**PostgresQL, мс**
1	76418	95213
5	328253	414436
10	663635	737435
25	1931132	2435168
100	7296236	9396236

В рисунках 7 - 9 представлены визулизация результатов поставленного эксперимента в виде графиков.

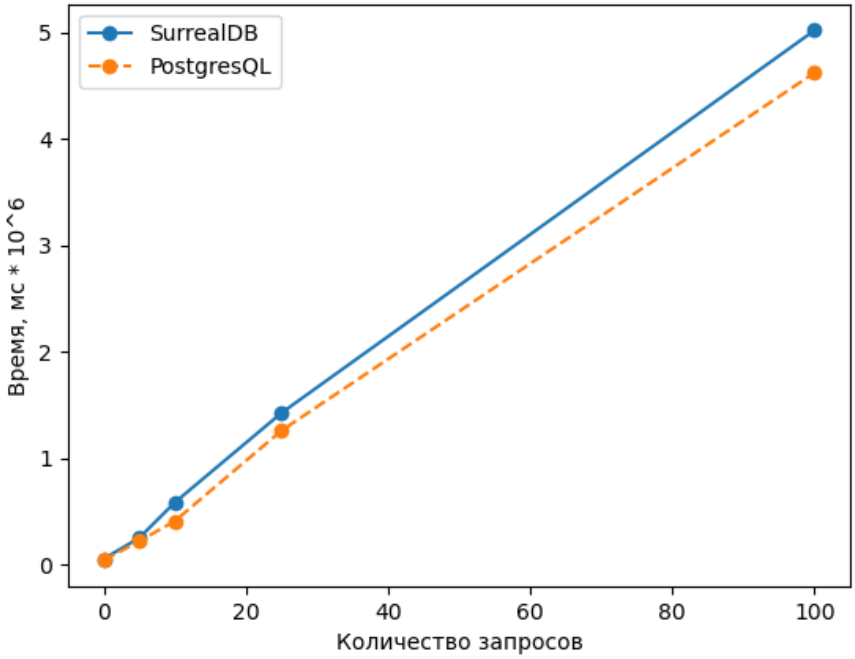

Рис. 7: Зависимость времени от количества запросов (количество элементов — 100)

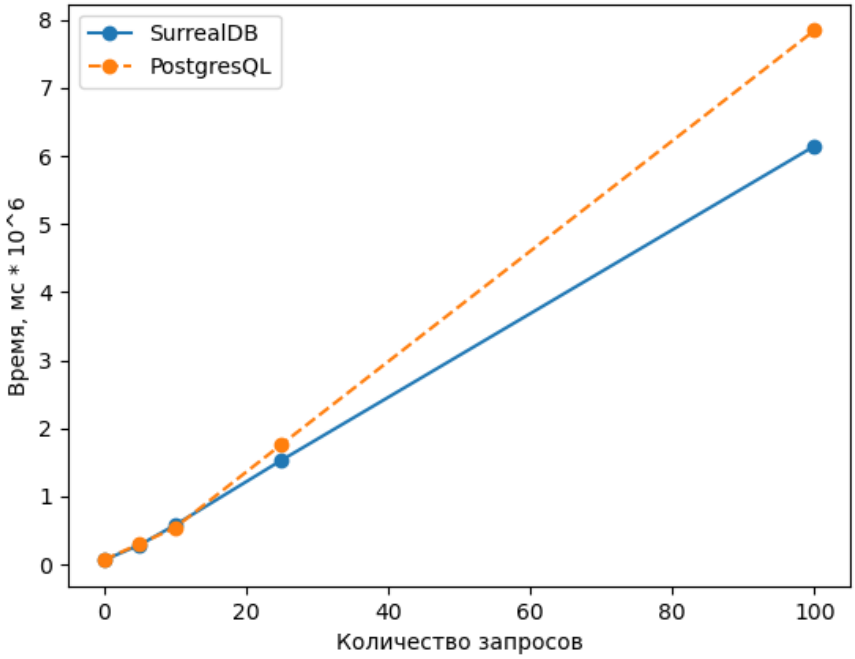

Рис. 8: Зависимость времени от количества запросов (количество элементов — 1000)

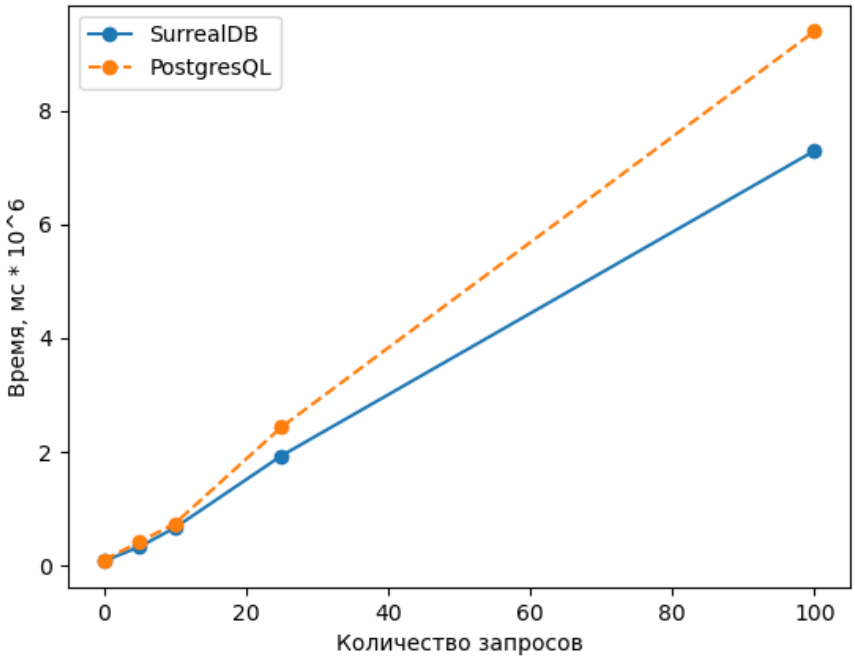

Рис. 9: Зависимость времени от количества запросов (количество элементов — 5000)

Вывод

Результаты эксперимента показали, что в то время как при небольшом количестве сущностей в базе данных `PostgresQL` показывает лучшие результаты, то при увеличении количества сущностей в базе данных `SurrealDB` показывает лучшие результаты. Из данного наблюдения можно сделать следующие выводы:

– Реляционные базы данных показывают лучшие результаты при небольшом количестве сущностей в базе данных или в случае, если данные не сильно связаны;

– Графовые базы данных показывают лучшие результаты при большом количестве сущностей в базе данных в случаях, если данные сильно связаны.

Даже несмотря на то, что в данном эксперименте были использованы только две базы данных, можно сделать вывод, что графовые базы данных показывают лучшие результаты при большом количестве сущностей в базе данных в случаях, если данные сильно связаны. Учитывая подобный выигрыш по времени графовых баз данных над реляционными, можно сделать вывод, что графовые базы данных являются более оптимальным решением для таких задач как: Социальные сети, рекомендательные системы, анализ связей в бизнесе и прочих задач, где основой данных являются связи между сущностями.

ЗАКЛЮЧЕНИЕ

В ходе выполнения проекта, цель данного курсовой работы была достигнута, то есть был разработан программный продукт, позволяющий эффективно хранить результаты тестов Тьюринга. В ходе выполнения экспериментально-исследовательской части было установлено, что запросы по связям будут выполняться быстрее в графовых базах данных, нежели запросы с использованием множественных JOIN-запросов в реляционных базах данных. При этом следует помнить, что значительные выигрыши по времени могут наблюдаться только при условии большого размера базы данных и сильной связанности сущностей внутри неё.

Для достижение цели были выполнены следующие задачи:

- проведен анализ предметной области — Тест Тьюринга;
- спроектирована архитектура программного обеспечения, на которой были отображены сущности и возможные связи между ними;
- выбраны средства реализации программного обеспечения с подробным описанием их особенностей;
- разработано программное обеспечение;
- разработаны и проведены эксперименты по замеру времени работы программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] What is natural language processing? [Электронный ресурс]. Режим доступа: https://www.ibm.com/topics/natural-language-processing.

[2] Bourbakis Nikolaos G. Artificial Intelligence and Automation. World Scientific, 1998.

[3] TURING A. M. I.—COMPUTING MACHINERY AND INTELLIGENCE // Mind. 1950. 10. Т. LIX, № 236. С. 433–460. URL: https://doi.org/10.1093/mind/LIX.236.433.

[4] Oppy Graham, Dowe David. The Turing Test // The Stanford Encyclopedia of Philosophy / под ред. Edward N. Zalta. Metaphysics Research Lab, Stanford University, 2021.

[5] Yoon Byoung-Ha, Kim Seon-Kyu, Kim Seon-Young. Use of graph database for the integration of heterogeneous biological data // Genomics & Informatics. 2017. Mar. Т. 15, № 1. с. 19–27.

[6] SurrealDB | Features [Электронный ресурс]. Режим доступа: https://surrealdb.com/features (дата обращения: 09.04.2023).

[7] SurrealDB | Architecture [Электронный ресурс]. Режим доступа: https://surrealdb.com/docs/introduction/architecture (дата обращения: 09.04.2023).

[8] TiKV | Overview [Электронный ресурс]. Режим доступа: https://tikv.org/ (дата обращения: 09.04.2023).

[9] Neo4j Graph Database & Analytics | Graph Database And Management System [Электронный ресурс]. Режим доступа: https://neo4j.com/ (дата обращения: 09.04.2023).

[10] Sikhinam Tharun. How does the performance of a graph database such as Neo4j compare to the performance of a relational database such as

Postgres? [Электронный ресурс]. Режим доступа: https://courses.cs.washington.edu/courses/csed516/20au/projects/p06.pdf (дата обращения: 09.04.2023).

[11] Electron | Documentation [Электронный ресурс]. Режим доступа: https://www.electronjs.org/docs/latest/ (дата обращения: 12.08.2023).

[12] Rust [Электронный ресурс]. Режим доступа: https://www.rust-lang.org/. Дата обращения: 19.08.2023.

[13] Rust Borrow Checker [Электронный ресурс]. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[14] Tauri Architecture [Электронный ресурс]. Режим доступа: https://tauri.app/v1/references/architecture/ (Дата обращения: 19.08.2023).

[15] Arch Linux [Электронный ресурс]. Режим доступа: https://archlinux.org/. Дата обращения: 10.09.2023.

[16] Процессор Intel® Core ™ i5-11320H [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu.html. Дата обращения: 19.10.2022.

ПРИЛОЖЕНИЕ А

Презентация к курсовой работе

Презентация содержит 13 слайдов.