МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Институт (Филиал) № 8 «Компьютерные науки и прикладная математика» **Кафедра** 806

Группа М8О-406Б-21 **Направление подготовки** 01.03.02 «Прикладная математика и информатика»

Профиль Информатика

Квалификация: бакалавр

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

на тему: Проектирование системы переноса и генерации взаимосвязанных данных из производственной среды при тестировании образовательной платформы

Автор ВКРБ:	Мезенин Олег Александрович	(_____)
Руководитель:	Миронов Евгений Сергеевич	(_____)
Консультант:	Ляпина Светлана Юрьевна	(_____)
Консультант:	—	(_____)
Рецензент:	—	(_____)

К защите допустить

Заведующий кафедрой № 806 Крылов Сергей Сергеевич (_____)

____ мая 2025 года

Москва 2025

РЕФЕРАТ

Выпускная квалификационная работа бакалавра состоит из 61 страницы, 34 рисунков, 1 таблицы, 44 использованных источников, 1 приложения.

БАЗЫ ДАННЫХ, ПЕРЕНОС ВЗАИМОСВЯЗАННЫХ ДАННЫХ, ТЕСТИРОВАНИЕ ОБРАЗОВАТЕЛЬНОЙ ПЛАТФОРМЫ, POSTGRESQL, ГЕНЕРАЦИЯ ДАННЫХ

В выпускной квалификационной работе бакалавра была спроектирована система переноса и генерации взаимосвязанных данных при тестировании образовательной платформы. Разработаны алгоритмы переноса и генерации взаимосвязанных данных, создана грамматика языка для описания входных данных. Был разработан и протестирован прототип системы, позволяющий переносить данные, и был проведён анализ жизнеспособности системы.

Систему можно использовать при тестировании программных продуктов любой сферы деятельности.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящей выпускной квалификационной работе бакалавра применяют следующие термины с соответствующими определениями:

Надёжность системы — способность системы продолжать работу в случае сбоев, минимизировать время простоя

Безопасность системы — способность системы обеспечивать защиту информации от несанкционированного доступа

Качество программного обеспечения — комплекс характеристик программного обеспечения, определяющих его способность выполнять функции в соответствии с заданными требованиями

Перенос данных — это процесс копирования и перемещения данных из одного источника или системы в другую без удаления данных из оригинальной базы данных

CI/CD — сокращение от Continuous Integration и Continuous Deployment, представляет собой совокупность практик и инструментов, применяемых в области разработки программного обеспечения с целью автоматизации и оптимизации процессов интеграции, тестирования и развёртывания приложений

DevOps-инженер — это специалист, работающий на стыке разработки программного обеспечения (development) и его эксплуатации (operations), задачей которого является создание надёжного и эффективного процесса разработки, тестирования и развёртывания приложений

CLI — сокращение от Command Line Interface, представляет собой интерфейс взаимодействия пользователя с операционной системой или программным обеспечением, который осуществляется посредством ввода текстовых команд в терминал

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей выпускной квалификационной работе бакалавра применяют следующие сокращения и обозначения:

БД — база данных

СУБД — система управления базами данных

ПО — программное обеспечение

ВВЕДЕНИЕ

В настоящее время рынок онлайн-образования демонстрирует значительный рост: объём российского рынка образовательных технологий в 2024 году составил 149 миллиардов рублей, увеличившись на 21% по сравнению с 2023 годом [1]. Сегмент онлайн-образования, направленный на детей, составляет 29% [2]. В связи с этим внимание следует уделить платформам, предназначенным для участников средней общеобразовательной школы.

Эти платформы требуют надёжной системы управления данными для обеспечения стабильной работы и поддержки большого количества пользователей. Одним из распространённых решений в области управления базами данных является PostgreSQL [3] — реляционная система управления базами данных с открытым исходным кодом.

В процессе разработки платформы важно уделять особое внимание качеству и надёжности. Это делает тестирование незаменимой стадией разработки. Благодаря тестированию можно убедиться в стабильности и бесперебойной работе систем.

Использование реальных данных и проведение тестов в производственной среде позволяет максимально точно оценить работоспособность системы. Но такой подход сопряжён с рисками, такими как нарушения конфиденциальности данных и потенциальные сбои в работе системы.

Чтобы тестирование образовательных платформ было безопасным и в то же время позволяло бы максимально точно оценить работоспособность системы, необходимо создать тестовую среду, максимально приближенную к производственным условиям. Создание тестовой среды требует тщательной проработки и учёта всех аспектов, связанных с функционированием платформы. Это включает в себя настройку инфраструктуры, идентичной производственной, и воспроизведение всех взаимосвязей данных, которые присутствуют в реальных условиях эксплуатации.

Особую значимость в этом контексте приобретает работа с данными. Для достижения максимальной схожести с производственной средой необходимо использовать тестовые данные, которые точно отражают объёмы, структуры и взаимосвязи данных, присутствующие в реальной системе.

Создание тестового набора данных может быть реализовано с использованием различных методик. Один из подходов включает полный перенос данных из производственной базы данных с последующей анонимизацией конфиденциальной информации. Утилита pg_dump [4] может быть использована для экспорта данных из PostgreSQL, обеспечивая перенос структуры и содержимого базы данных.

Для защиты пользовательской информации и анонимизации данных можно применить утилиту pg_anonymizer [5], которая помогает скрыть чувствительные данные, заменяя их на фиктивные значения, что сохраняет конфиденциальность пользователей.

Основной недостаток данного метода заключается в невысокой скорости переноса всех данных, что может быть критичным при работе с крупными наборами данных и требовать значительных временных затрат для выполнения всей операции.

Частичный перенос данных может быть более подходящим для тестирования, так как зачастую не требуется полный объём данных для покрытия большинства сценариев. Но этот подход сопряжён с рядом вопросов и потенциальных сложностей, которые опишем ниже.

При полном переносе данных тестировщик получает доступ ко всем данным, что позволяет ему проводить тестовые сценарии. В случае частичного переноса данных требуется определить и описать необходимые для тестирования данные, избегая глубокого изучения структуры базы данных и её связей. Возникает вопрос, каким образом тестировщику описать данные, нужные для проведения тестов.

Следующим вопросом является сохранение целостности данных. В случае полного переноса реляционная целостность поддерживается автоматически, поскольку все взаимосвязанные данные переносятся полностью, обеспечивая корректность и согласованность системы. При частичном переносе необходимо уделить особое внимание поддержанию целостности данных. Недопустимо, чтобы после переноса в тестовой среде данных было недостаточно для выполнения тестовых сценариев. Следовательно, второй вопрос заключается в обеспечении целостности данных при минимизации объёма переносимых данных.

Перенос данных из производственной среды с использованием анонимизации представляет собой лишь один из способов получения

тестовых данных. Альтернативный подход состоит в генерации данных на основе заданных характеристик. Этот процесс может выглядеть следующим образом. Пользователь задаёт параметры, которые могут сопровождаться набором или диапазоном значений. Затем программа генерирует тестовые данные, основываясь на фактических данных, либо на схеме данных производственной базы, а также на указанных пользователем характеристиках. Таким образом, полученные данные будут иметь структуру, аналогичную реальным данным, но не будут связаны с конкретными записями в производственной базе. Но для этого метода возникают вопросы описания пользователем необходимых данных и их корректной генерации.

Дипломная работа будет посвящена проектированию системы, которая должна поддерживать следующую функциональность:

- перенос взаимосвязанных данных из одной базы данных в другую с применением методов анонимизации,
- генерацию тестовых данных.

В рамках работы будут рассмотрены алгоритмы переноса и генерации данных. Также будет разработан язык для описания данных и прототип системы.

1 Постановка задачи и теоретические предпосылки

1.1 Тестирование

Для определения требований искомой системы необходимо формализовать процесс тестирования программного обеспечения.

1.1.1 Процесс тестирования

Тестирование программного обеспечения – это процесс проверки и оценки работы программы или системы для выявления ошибок, дефектов и несоответствий. Основная цель тестирования – удостовериться в том, что ПО работает корректно и удовлетворяет требованиям пользователей [6].

Процесс тестирования можно разбить на несколько ключевых этапов [7]:

а) планирование:
- разработка стратегии тестирования, включая выбор типов тестирования,
- создание тест-плана, где описываются ресурсы, сроки и критерии начала и завершения тестирования,
- проведение анализа рисков, связанных с тестированием, и определение действий для минимизации их воздействия,

б) подготовка:
- разработка тестовых случаев и сценариев на основе требований и спецификаций,
- подготовка тестовой среды, включая оборудование и программные средства, необходимые для проведения тестов,
- подготовка данных для тестирования и настройка автоматизированных тестов, если они предусмотрены,

в) проведение:
- непосредственное выполнение тестов согласно плану тестирования,
- фиксация и документирование результатов тестирования, выявление дефектов и несоответствий,

г) совершенствование:
- анализ результатов тестирования и составление отчёта по

итогам выполненной работы,

- обмен опытом и полученными знаниями внутри команды и с другими заинтересованными сторонами,
- обновление и улучшение тестовых документов и процессов на основе полученных данных.

Наша система будет применяться на этапе подготовки: с её помощью можно будет создавать тестовые данные.

1.1.2 Ручное и автоматизированное тестирование

Тестирование можно разделить на ручное и автоматизированное [8].

Ручное тестирование предполагает процесс, при котором тестировщик выполняет тесты без помощи автоматизированных инструментов. Оно позволяет глубже вникнуть в пользовательский опыт, и его легко применять к новым или часто изменяющимся функциональностям. Но ручное тестирование может занимать много времени, оно подвержено человеческим ошибкам и не всегда позволяет повторить результаты для их сравнения.

Автоматизированное тестирование предполагает использование программных инструментов для автоматизации выполнения тестовых сценариев. Это предполагает разработку скриптов, которые автоматически выполняют тесты и сверяют результаты с ожидаемыми. Автоматизированное тестирование обеспечивает высокую скорость и стабильность результатов, что делает его эффективным для регрессионного тестирования и стабильных функциональностей. Но автоматизированное тестирование требует значительных ресурсов для разработки скриптов и не всегда подходит для тестирования пользовательских интерфейсов и динамически изменяющихся требований.

Оба подхода часто используются совместно для достижения наилучшего результата. Обычно ручное тестирование применяют на начальных стадиях или для исследовательского тестирования, тогда как автоматизированное тестирование использует для повторяющихся задач или сложных сценариев, которые требуют стабильного выполнения.

1.2 Сценарии использования

Перед определением требований к нашей системы рассмотрим

возможные сценарии использования.

Определим роли:

– тестировщик – специалист, занимающийся проверкой качества ПО,
– разработчик – специалист, занимающийся разработкой ПО,
– DevOps-инженер – специалист, занимающийся поддержкой процессов CI/CD.

Будем называть **метаданными** входные данные, включающие в себя правила переноса, генерации или анонимизации данных.

1.2.1 Перенос данных для проведения тестовых сценариев

Роли:

– тестировщик,
– разработчик.

Цель: создать тестовые данные на основе существующих реальных данных для их дальнейшего применения в тестовых сценариях.

Предпосылки:

– разработчиком написаны метаданные, включающие в себя правила переноса и анонимизации данных,
– тестировщик осведомлен о метаданных, необходимых для проведения тестовых сценариев,
– тестировщик имеет уникальный идентификатор исходной базы данных,
– имеется пустая тестовая база данных, доступ к которой имеется у тестировщика.

Основной сценарий:

а) тестировщик осуществляет изменения в метаданных,
б) тестировщик инициирует запуск CLI, указывая следующие параметры:
    – уникальный идентификатор исходной базы,
    – уникальный идентификатор целевой базы или параметры подключения к ней,
    – метаданные или уникальный идентификатор метаданных,
в) тестировщик ожидает завершения процесса переноса данных.

Постусловия:

– тестировщик применяет данные, сгенерированные на основе

реальных данных, в тестовых сценариях.

Исключения:

– тестировщик получает ошибку некорректных метаданных после пункта б.

1.2.2 Диагностика ошибки

Роли:

– тестировщик.

Цель: провести диагностику ошибки, возникшей с определенными данными в продуктивной среде.

Предпосылки:

– в продуктивной среде обнаружены проблемы с определенными данными, такими как данные пользователя,
– тестировщик владеет уникальным идентификатором данных, в которых выявлена ошибка в продуктивной среде,
– тестировщик имеет уникальный идентификатор исходной базы данных,
– имеется пустая тестовая база данных, доступ к которой имеется у тестировщика.

Основной сценарий:

а) тестировщик описывает метаданные, включая данные, в которых произошла ошибка,
б) тестировщик инициирует запуск CLI, указывая следующие параметры:
    – уникальный идентификатор исходной базы,
    – уникальный идентификатор целевой базы или параметры подключения к ней,
    – метаданные,
в) тестировщик ожидает завершения процесса переноса данных.

Альтернативный сценарий:

– тестировщик может взять готовые метаданные, если они ему подходит, а не описывать свои.

Постусловия:

– тестировщик осуществляет взаимодействие с перенесенными данными и проводит диагностику ошибки.

Исключения:

– тестировщик получает ошибку некорректных метаданных после пункта б.

1.2.3 Тестирование новой миграции

Роли:

– тестировщик,
– разработчик.

Цель: провести тестирование новой миграции.

Предпосылки:

– тестировщик имеет уникальный идентификатор исходной базы данных,
– имеется пустая тестовая база данных, доступ к которой имеется у тестировщика,
– разработчик создал новую миграцию, нуждающуюся в тестировании,
– разработчик предоставил метаданные к миграции, содержащие правила анонимизации и переноса данных.

Основной сценарий:

а) тестировщик инициирует запуск CLI, указывая следующие параметры:
    – уникальный идентификатор исходной базы,
    – уникальный идентификатор целевой базы или параметры подключения к ней,
    – предоставленные метаданные,
б) тестировщик ожидает завершения процесса переноса данных.

Постусловия:

– тестировщик взаимодействует с перенесенными данными и осуществляет тестирование миграций.

1.2.4 Перенос данных для автоматизированного тестирования

Роли:

– тестировщик,
– разработчик,

– DevOps-инженер.

Цель: обеспечеть возможность проведения автоматизированного тестирования изменений через сценарий CI/CD.

Предпосылки:

– разработчиком написаны метаданные, включающие в себя правила переноса и анонимизации данных,
– тестировщиком реализованы автоматизированные тесты,
– DevOps-инженер разработал сценарий CI/CD, который, в случае появления нового запроса на слияния в системе контроля версий, инициирует развёртывание тестового окружения с пустой базой данных и осуществляет перенос необходимых данных с помощью нашей системы.

Основной сценарий:

а) разработчик вносит изменения в исходный код и создает новый запрос на слияние в системе контроля версий,
б) запускается процесс CI/CD, который выполняет следующие действия:
    – развёртывание тестового окружения с пустой базой данных,
    – перенос необходимых данных в тестовую базу с помощью нашей системы,
    – автоматический запуск тестов для проверки корректности внесённых изменений.

1.2.5 Генерация данных для проведения тестовых сценариев

Роли:

– тестировщик,
– разработчик.

Цель: создать тестовые данные на основе существующих реальных данных для их дальнейшего применения в тестовых сценариях.

Предпосылки:

– разработчиком написаны метаданные, включающие в себя правила генерации данных,
– тестировщик осведомлен о метаданных, необходимых для проведения тестовых сценариев,

– тестировщик имеет уникальный идентификатор исходной базы данных,
– имеется пустая тестовая база данных, доступ к которой имеется у тестировщика.

Основной сценарий:

а) тестировщик осуществляет изменения в метаданных,
б) тестировщик инициирует запуск CLI, указывая следующие параметры:
    – уникальный идентификатор исходной базы,
    – уникальный идентификатор целевой базы или параметры подключения к ней,
    – метаданные или уникальный идентификатор метаданных,
в) тестировщик ожидает завершения процесса генерации данных.

Постусловия:

– тестировщик применяет сгенерированные данные в тестовых сценариях.

Исключения:

– тестировщик получает ошибку некорректных метаданных, после пункта б.

1.3 Определение требований к системе

Требования к системе можно разделить на две основные категории: функциональные и архитектурные свойства [9]. Функциональные требования описывают конкретный набор функций и возможностей, которые система должна обеспечивать для удовлетворения потребностей пользователей. Требования к свойствам архитектуры системы фокусируются на качественных характеристиках системы.

1.3.1 Функциональные требования

Из приведённых выше сценариев использования можно заключить, что система должна поддерживать следующую функциональность:

– перенос и анонимизация данных. Система должна обеспечивать возможность переноса взаимосвязанных данных между базами данных и их анонимизацию. Правила выбора взаимосвязанных

данных и правила анонимизации задаются метаданными;

- генерация данных. Система должна предоставлять функциональность генерации данных. Правила генерации задаются метаданными;
- работа с метаданными. Система должна осуществлять проверку корректности метаданных и обеспечивать функциональность для загрузки предварительно подготовленных метаданных или их интеграции на этапе ввода данных;
- гибкость в выборе базы данных. Необходимо внедрить функциональность, позволяющую пользователю выбрать базу данных, указав как уникальный идентификатор базы, так и строку подключения.

1.3.2 Требования к свойствам архитектуры

Существует множество архитектурных свойств [10].

Определим основные свойства, которым должна соответствовать система, в порядке приоритета:

- безопасность. Архитектура должна обеспечивать защиту от несанкционированного доступа к подключаемым базам данных. Обеспечение безопасности системы является самым приоритетным свойством, потому что ответственность особенно высока при работе с персональными данными, так как это требует соблюдения строгих правовых норм;
- производительность. Архитектура должна обеспечивать высокую скорость переноса и генерации взаимосвязанных данных. Данное требование обусловлено недостаточной эффективностью существующих решений в области переноса данных, что было обосновано в вводной части дипломной работы;
- надёжность. Система должна функционировать без сбоев в течение продолжительных периодов времени. Это предполагает наличие мер по обеспечению отказоустойчивости и возможности восстановления после сбоев. Несмотря на потенциальное превосходство в производительности по сравнению с существующими аналогами, процессы переноса и генерации данных могут характеризоваться значительной временной

продолжительностью. Прерывание данных процессов может привести к существенному замедлению процесса тестирования;

– асинхронная обработка запросов. Система должна поддерживать выполнение запросов в асинхронном режиме. Это означает, что клиентские приложения не должны блокироваться в ожидании завершения операции, но должны иметь возможность проверять статус выполнения запроса по мере необходимости.

2 Проектирование системы, разработка языка и алгоритма

2.1 Архитектура системы

Рассмотрим архитектуру системы по модели C4 [11].

2.1.1 Взаимодействие систем

В схеме архитектуры на уровне контекста можно наблюдать взаимодействие различных систем, а также взаимодействие этих систем с пользователями и специалистами. Схема взаимодействия представлена на рисунке 1.

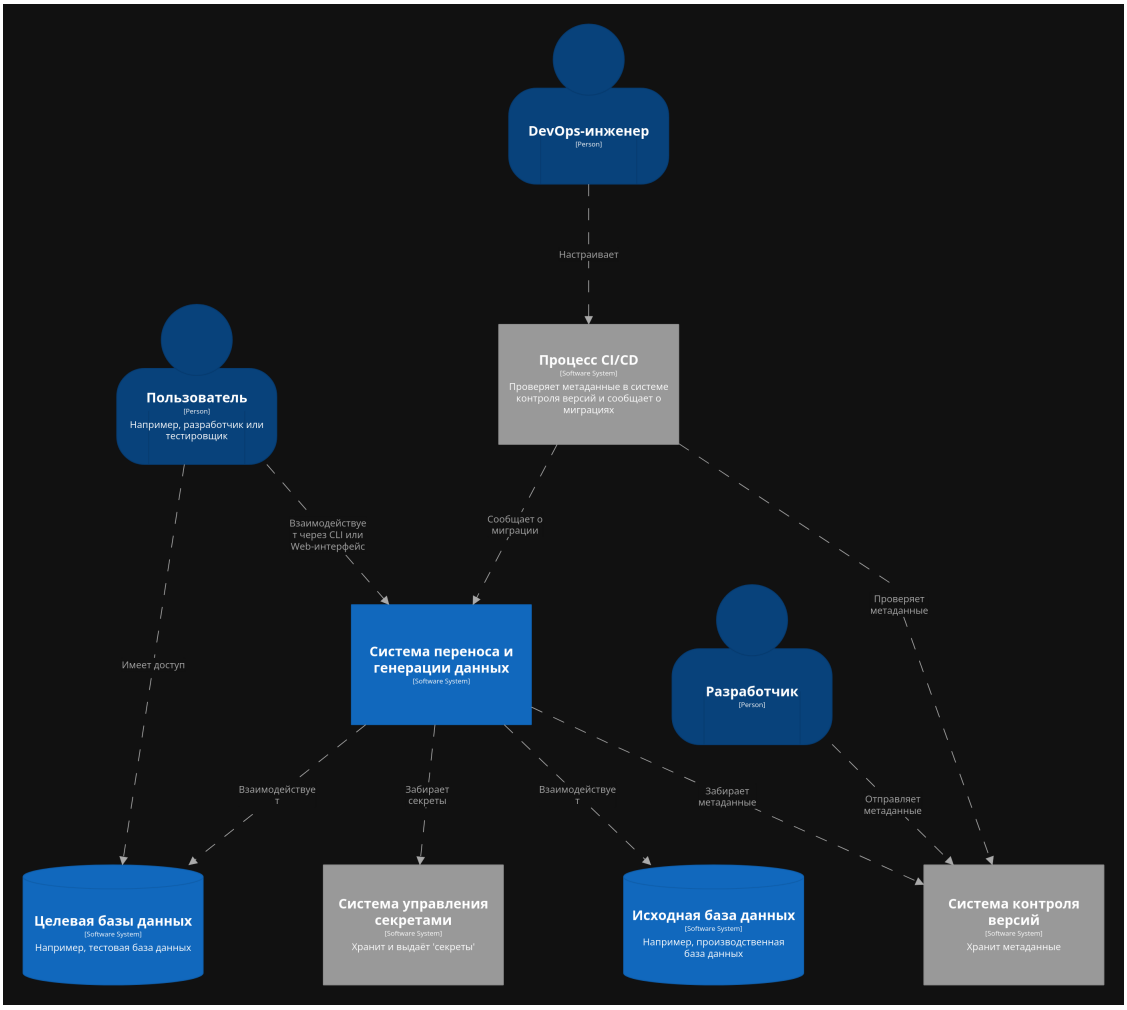

Рисунок 1 – Взаимодействие систем

Внешние системы выделены серым цветом:

– система контроля версий выступает в качестве хранилища метаданных, представленных разработчиком;
– система управления секретами предоставляет данные для

подключения к базе данных;

- процесс CI/CD, который настраивает DevOps-инженер, выполняет несколько функций:
  - генерация тестовой среды и запуск автоматизированных тестов при получении запроса на слияние в систему контроля версий;
  - проверка корректности метаданных в момент их фиксации в системе контроля версий;
  - при выполнении миграций осуществляется уведомление системы переноса и генерации данных о процессах миграции. Такой механизм необходим для предотвращения неопределенного поведения, которое может возникнуть, если система в текущий момент работает с мигрируемой базой данных.

Синим цветом выделены собственные системы: исходная и целевая базы данных, с которыми осуществляется взаимодействие пользователей, а также система переноса и генерации данных, которую мы рассмотрим более детально далее.

2.1.2 Система переноса и генерации данных

Рассмотрим систему переноса и генерации данных на уровне контейнеров. Схема контейнеров представлена на рисунке 2.

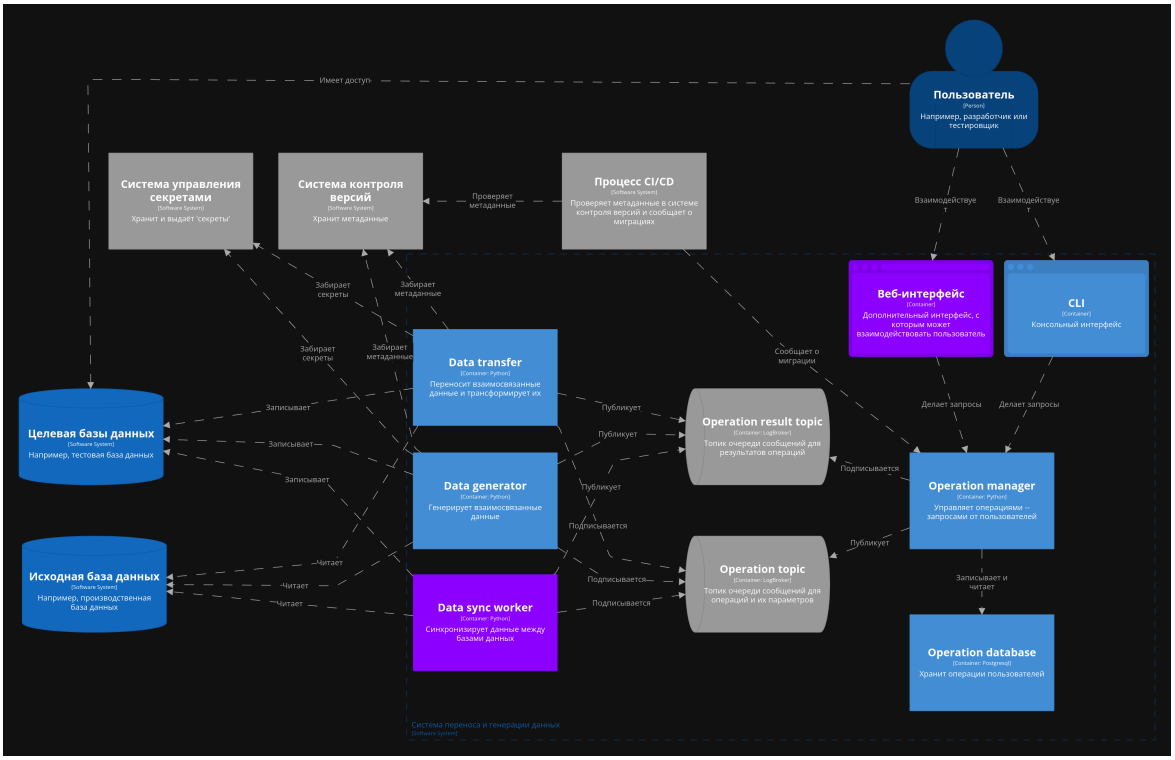

Рисунок 2 – Система переноса и генерации данных

Взаимодействие пользователя с системой осуществляется через командную строку (CLI), хотя может быть реализован и альтернативный интерфейс взаимодействия, например, веб-интерфейс (выделен фиолетовым как контейнер, который может быть добавлен в перспективе).

Пользователь посылает запросы в контейнер Operation Manager. Возможны два типа запросов: запуск операции по переносу или генерации данных, а также проверка статуса выполняемой операции. В случае получения запроса на запуск операции, информация об операции сохраняется в базе данных, а уникальный идентификатор операции возвращается пользователю, что позволяет ему отслеживать статус выполнения.

Далее Operation Manager отправляет запросы на выполнение операции в очередь сообщений Logbroker [12]. Контейнер, обрабатывающий такой запрос, определяется типом операции: если речь идет о переносе данных, операция обрабатывается контейнером Data Transfer; если о генерации данных — контейнером Data Generator.

На схеме также представлен контейнер Data Sync Worker, предназначенным для обработки операций по синхронизации данных в базах данных. Более подробно этот контейнер в данной работе рассмотрен не будет.

Контейнеры Data Transfer и Data Generator осуществляют перенос и

генерацию данных, взаимодействуя с базами данных, системой контроля версий и системой управления секретами. В процессе выполнения операции, а также после её выполнения, информация о статусе возвращается в очередь сообщений, из которой Operation Manager извлекает данные и обновляет статус операции в базе данных.

Более детальное описание структуры компонентов Data Transfer и Data Generator будет представлено в последующих разделах.

Рассмотрим диаграмму последовательности взаимодействия пользователя с системой на примере генерации данных. Диаграмма представлена на рисунке 3. На диаграмме можно заметить, как клиент и система общаются асинхронно. Это функциональность закрывает требование **асинхронной обработки запросов** системы.

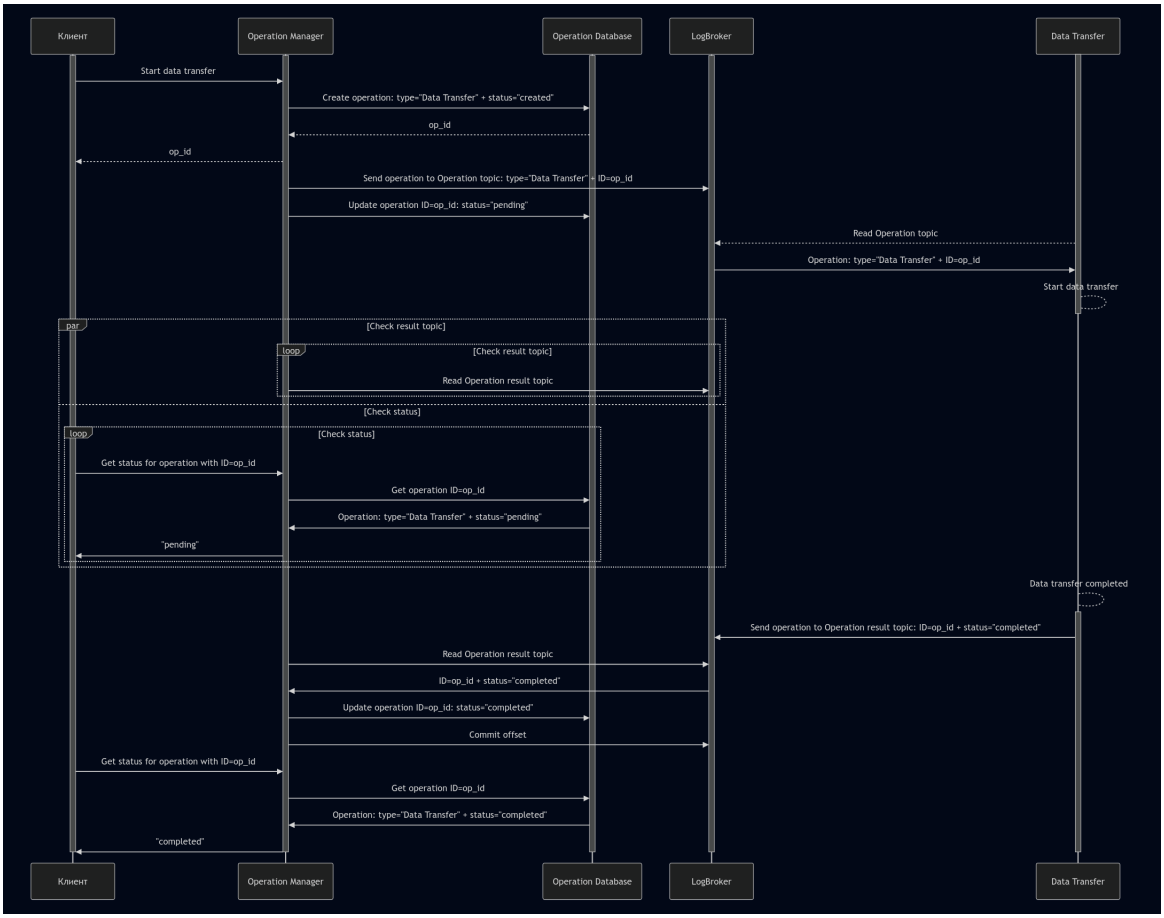

Рисунок 3 – Диаграмма последовательности взаимодействия пользователя с системой на примере генерации данных

2.1.3 Компоненты Data Transfer

Рассмотрим процесс обработки запросов на перенос задач. Схема

компонентов контейнера Data Transfer представлена на рисунке 4.

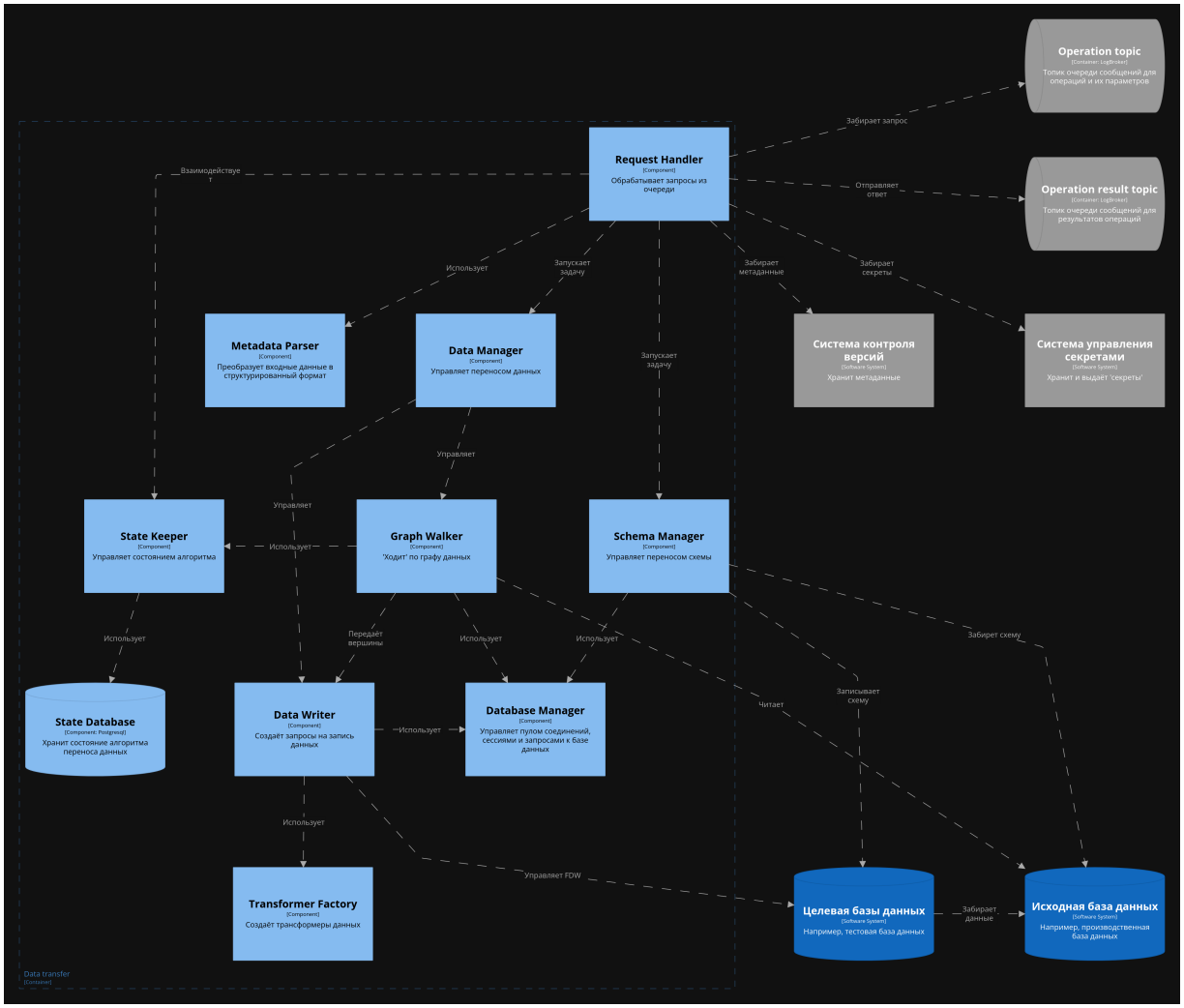

Рисунок 4 – Компоненты Data Transfer

Компонент Request Handler принимает такие запросы. Они должны включать следующие элементы:

– персональный токен доступа пользователя, который присылает запрос,
– идентификатор метаданных,
– идентификатор исходной базы данных,
– идентификатор целевой базы данных.

Идентификатор метаданных может принимать форму одной из двух сущностей: уникального идентификатора, связанного с метаданными в Системе контроля версий, или самих метаданных. В случае предоставления уникального идентификатора, Request Handler обращается к Системе контроля версий для извлечения соответствующих метаданных.

Request Handler осуществляет процедуру верификации прав доступа

путем сопоставления идентификаторов баз данных с персональным токеном доступа пользователя, инициировавшего запрос [13]. В процессе верификации производится проверка полномочий доступа к исходной и целевой базам данных. В случае отсутствия разрешений у пользователя, дальнейшая обработка запроса прекращается. Такой механизм закрывает требование **безопасности** системы.

Идентификатор базы данных может быть представлен либо в форме уникального идентификатора кластера базы данных, либо в виде параметров подключения к базе данных. При предоставлении уникального идентификатора кластера базы данных, Request Handler обращается к Системе управления секретами для получения параметров подключения, включая пароль.

Полученные метаданные направляются в компонент Metadata Parser для их валидации и преобразования во внутренний формат представления. Далее инициируется Data Manager, который получает информацию о подключениях к базам данных, а также внутреннее представление метаданных, и запускает компоненты Graph Walker и Data Writer.

Компонент Graph Walker выполняет алгоритм обхода данных исходной базы, который будет рассмотрен позже, следуя инструкциям, заданным в метаданных. Он использует State Keeper для сохранения состояния алгоритма и Database Manager для выполнения операций с базой данных. В процессе работы, Graph Walker извлекает метаинформацию о посещенных вершинах и асинхронно передает её компоненту Data Writer.

Компонент Data Writer управляет преобразованием данных и их записью в целевую базу данных. Он принимает на вход метаинформацию о данных, включая их идентификаторы, и с помощью Database Manager создаёт объект Foreign Data Writer [14] в целевой СУБД, обеспечивая прямое подключение к исходной СУБД. Data Writer формирует запросы, включающие идентификаторы данных, направляя их в целевую СУБД для извлечения конкретных данных из исходной базы. Запросы дополняются трансформерами, сформированными на основе метаданных через компонент Transformer Factory. Трансформер представляет собой набор правил преобразования данных, предназначенных для анонимизации данных.

Рассмотрим взаимодействие основных компонент на диаграмме последовательности, представленной на рисунке 5.

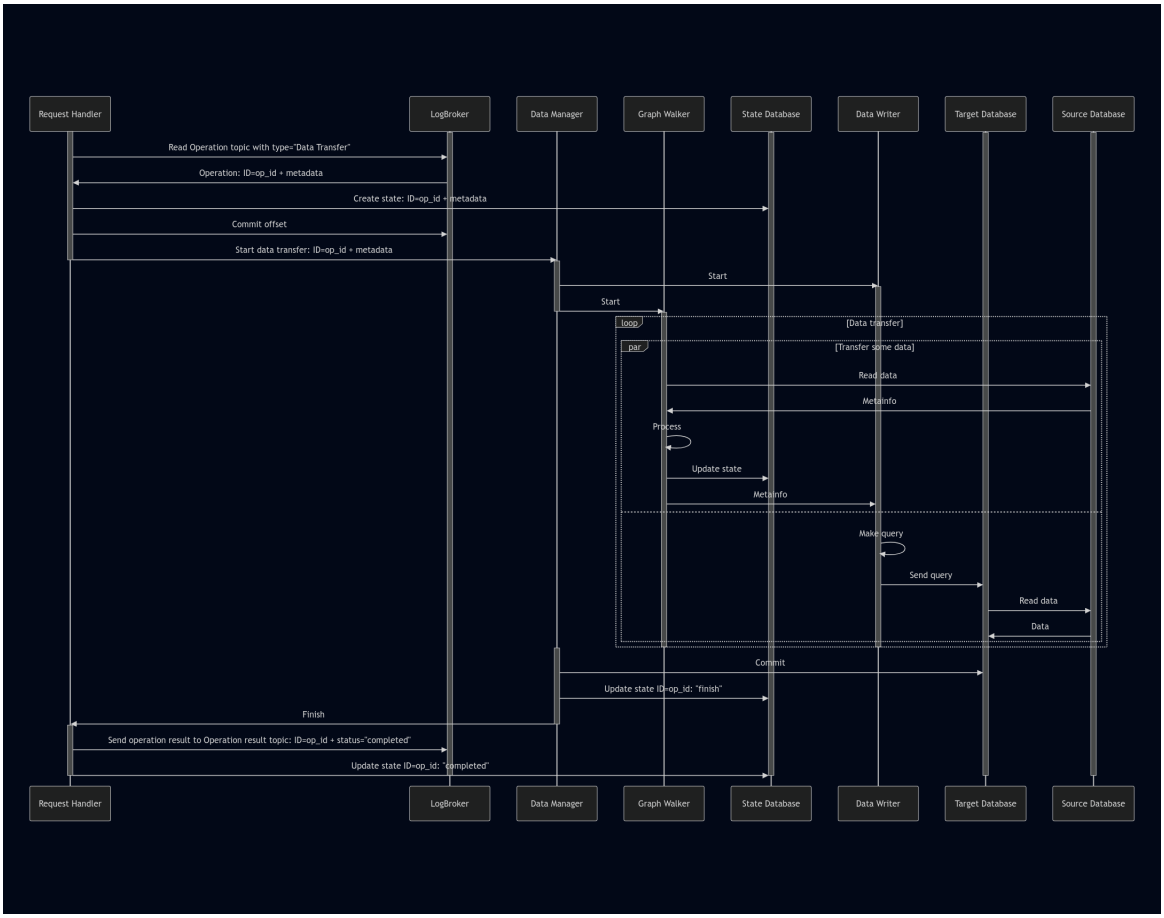

Рисунок 5 – Диаграмма последовательности взаимодействия пользователя с системой на примере генерации данных

Данный подход предусматривает непосредственный перенос данных из исходной базы в целевую, с возможными изменениями в соответствии с правилами, определёнными в метаданных, что способствует повышению **производительности** системы.

Также на рисунке 4 представлен компонент Schema Manager, который предоставляет функциональность по управлению схемами баз данных. В частности, данный компонент позволяет осуществлять перенос схемы между различными базами данных и выполнять удаление схемы.

2.1.4 Персистентность состояния и восстановление после сбоя

Процессы переноса данных могут быть времязатратными, независимо от объёма данных. В случае неожиданной остановки сервиса переноса и генерации данных, вызванной сбоем, утрата прогресса по переносу данных становится крайне нежелательным исходом.

Рассмотрим, как выполняется персистентность состояния системы, то

есть её способность сохранять и восстанавливать своё состояние после остановки.

Весь процесс выполнения операции от её инициации до её завершения можно разделить на два этапа:

– отправка и чтение из очереди сообщений,
– непосредственный процесс переноса данных.

Сбои могут возникать на любом из этих этапов. Для этапа отправки и чтения сообщений из очереди предусмотрена семантика доставки exactly-once [15], которая поддерживается использованием архитектурных паттернов Inbox Pattern и Outbox Pattern [16]. Эти паттерны, часть реализации которых были отражены на рисунках 3 и 5, обеспечивают гарантированную доставку сообщений и идемпотентную обработку входящих данных.

При сбое на этапе переноса данных механизмы восстановления включают проверку состояния незавершённых операций в базе данных состояния (State Database). В ней хранится прогресс каждой операции переноса данных и включает уникальные идентификаторы операции, состояние алгоритма и метаданные. При восстановлении система переноса данных проверяет наличие незавершённых операций и продолжает их с того места, где процесс был остановлен. Запись в целевую базу данных успешно завершается только при успешном выполнении алгоритма, что требует повторной записи данных в случае восстановления после сбоя, основываясь на метаинформации из сохранённого состояния.

Таким образом, архитектура системы включает механизмы обеспечения персистентности состояния и восстановления после сбоев, что закрывает требование **надёжности** системы.

2.2 Алгоритм обхода данных

Всю функциональность переноса данных можно разделить на две части: запись данных в целевую базу данных, которую рассматривали ранее, и обход исходных данных, алгоритм которого мы рассмотрим подробнее в этом разделе.

2.2.1 Метаграфы

На сегодняшний день в научном сообществе не существует общепринятого и устойчивого определения метаграфа. Работа [17] была

первым источником, в котором появился термин ”метаграф”. Эта концепция всё ещё находится в стадии развития и интерпретируется различными исследователями по-разному в зависимости от их специфических задач и целей [18; 19; 20; 21]. В общем смысле, метаграф можно рассматривать как структуру, которая обобщает и расширяет идеи классических графов, чтобы решить более сложные задачи анализа данных.

В данной дипломной работе предлагается собственное определение метаграфа, адаптированное под требования и задачи рассматриваемого алгоритма.

Пусть $MG = \langle V, MV, E, ME \rangle$ – метаграф, где V – множество вершин, MV – множество метавершин, E – множество рёбер, ME – множество метарёбер.

$v_i = \{atr_k\}, v_i \in V$, где v_i – вершина, atr_k – атрибут.

$mv_i = \langle \{v_j\}, \{atr_k\} \rangle, mv_i \in MV, v_j \in V$, где mv_i – метавершина, v_j – вершина, atr_k – атрибут.

$e_i = \langle v_s, v_e \rangle, e_i \in E, v_s, v_e \in V$, где e_i – ребро, v_s – исходная вершина, v_e – конечная вершина. $me_i = \langle mv_s, mv_e, \{atr_k\} \rangle, me_i \in ME, mv_s, mv_e \in MV$, где me_i – метаребро, mv_s – исходная метавершина, mv_e – конечная метавершина, atr_k – атрибут.

Также введём ограничение на множество метавершин: $\forall mv_i, mv_j \in V, i \neq j => \{v_x\}_i \cap \{v_y\}_j = \emptyset$, т.е. каждая вершина не может содержаться в нескольких метавершинах.

Следует отметить, что метаграф может быть **симметричным**. Это означает, что, во-первых, при наличии метаребра, соединяющего одну метавершину с другой, существует и обратное метаребро, соединяющее вторую метавершину с первой; во-вторых, наличие ребра между двумя обычными вершинами также подразумевает существование обратного ребра. Симметричный метаграф $MG = \langle V, MV, E, ME \rangle$ определяется следующим образом: если $\langle mv_i, mv_j, \{atr_k\} \rangle \in ME$, то также выполняется $\langle mv_j, mv_i, \{atr_k\} \rangle \in ME$, и если $\langle v_i, v_j \rangle \in E$, то также существует $\langle v_j, v_i \rangle \in E$.

2.2.2 Представление базы данных в виде метаграфа

Реляционные базы данных традиционно состоят из множества связанных таблиц, каждая из которых содержит структурированные данные. В метаграфе эти элементы могут быть представлены следующим образом.

– метавершины. В контексте метаграфа метавершинам соответствуют конкретные таблицы реляционной базы данных. Набор атрибутов метавершины содержат информацию о структуре таблицы, включая её название, идентификатор и набор столбцов;
– вершины. Вершины соответствуют записям в таблицах. Каждый набор атрибутов вершины напрямую соответствует набору значений в конкретной записи таблицы;
– метарёбра. Метарёбра описывают связи между различными таблицами в базе данных. Эти связи формируются посредством внешних ключей [22], которые указывают на зависимость одной таблицы от другой. Атрибуты метарёбер содержат информацию о полях, которые связаны между таблицами;
– рёбра. Рёбра представляют связи между конкретными записями в таблицах и формируются на основании соответствующих метарёбер. Эти рёбра конкретизируют связь на уровне данных, отражая указанные в метарёбрах отношения через конкретные внутренние связи записей, например, через одинаковые или согласованные идентификаторы.

Для иллюстрации представления базы данных в виде метаграфа рассмотрим небольшой пример, содержащий пять таблиц по несколько записей в каждой. На рисунке 6 представлена схема взаимосвязей между конкретными записями в базе данных. Красные стрелки указывают на то, что записи, из которых исходят стрелки, имеют ссылки на другие записи. Это свидетельствует о наличии в таблице внешнего ключа, ссылающегося на другую таблицу. В свою очередь, зелёные стрелки обозначают противоположную ситуацию: данные, на которые они указывают, являются объектом ссылок со стороны других записей.

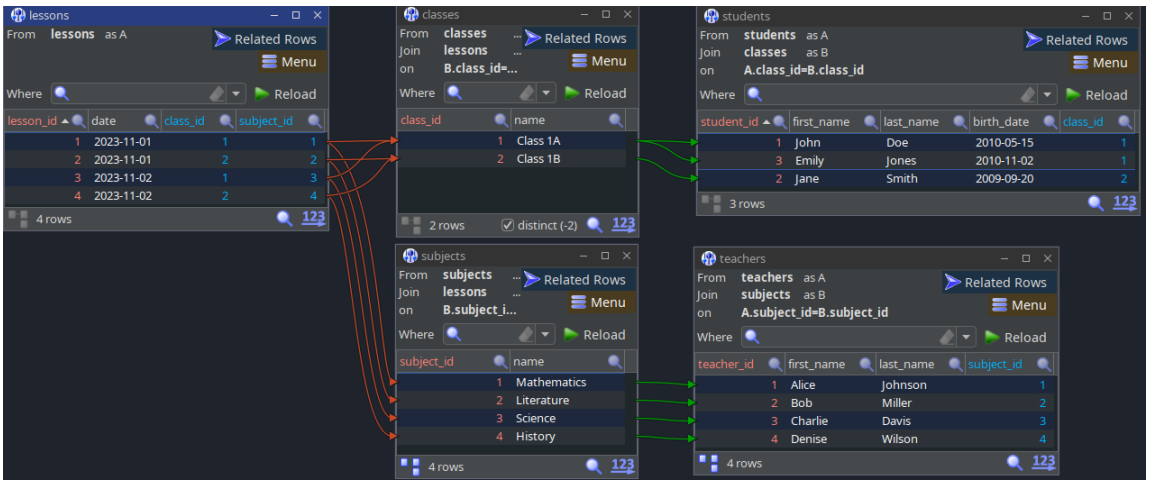

Рисунок 6 – Пример реляционной базы данных

На рисунке 7 представлено графическое изображение метаграфа, соответствующего заданной базе данных. Оранжевым цветом обозначены метавершины и метарёбра, которые соответствуют таблицам и связям между этими таблицами. В свою очередь, синим цветом представлены вершины и рёбра, отражающие записи и связи между ними.

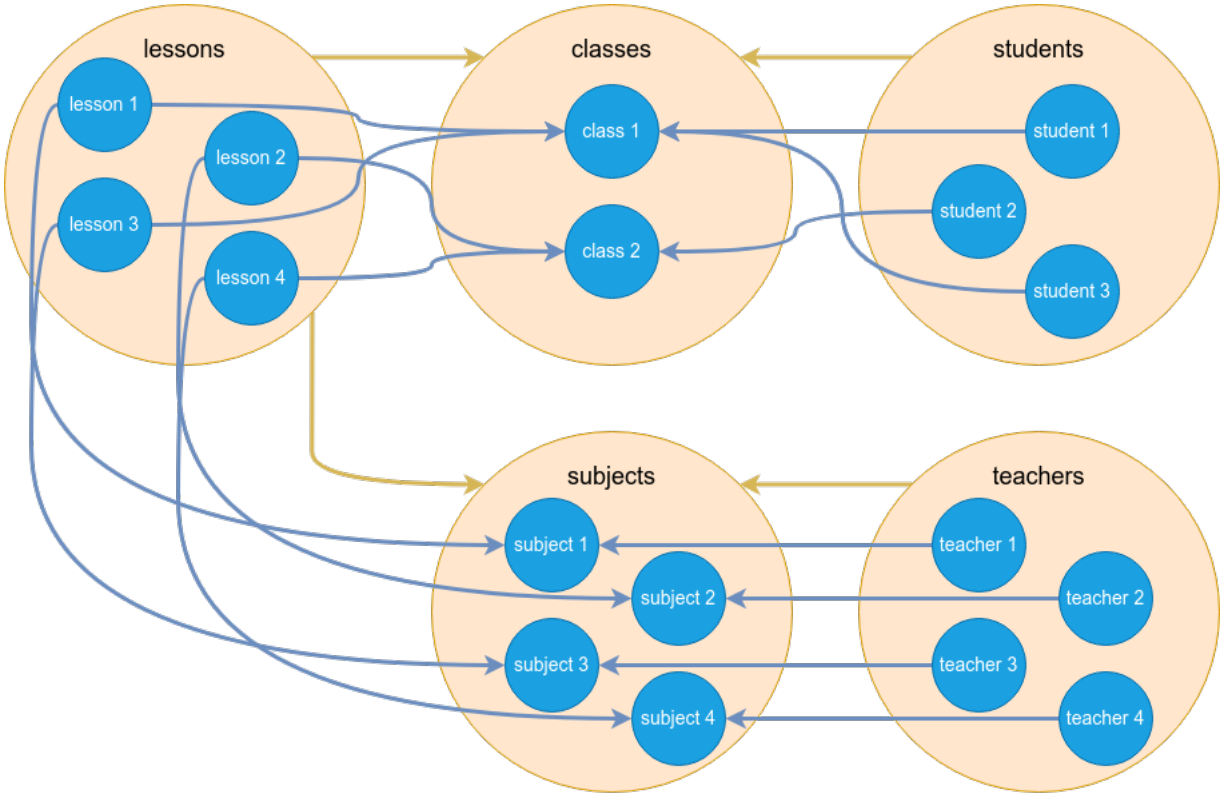

Рисунок 7 – Пример метаграфа

Далее, для краткости, такой метаграф будем называть **метаграфом базы данных**.

2.2.3 Описание алгоритма с метаграфом базы данных

Поскольку теперь можно представить базу данных в виде метаграфа, алгоритм обхода взаимосвязанных данных фактически будет являться алгоритмом обхода графа.

Такой алгоритм должен принимать на вход множество начальных вершин, которые соответствуют конкретным данным, а также метаграф базы данных. На выходе алгоритм должен предоставлять множество всех вершин, связанных с начальными, включая сами начальные вершины. После получения множества связанных вершин, появляется возможность извлечь из него конкретные данные.

В качестве основы алгоритма для решения поставленной задачи предлагается использовать метод поиска в ширину [23]. Данный подход позволяет эффективно обрабатывать структуры данных, представленные в виде графов. Алгоритм представлен на рисунке 8. Следует отметить, что псевдокод алгоритма разработан на основе соглашений и обозначений, представленных в работе [24].

```
1  Base_Metagraph_Traversal(DB, SV)
2    queue ← ∅
3    visited ← ∅
4    for each v in SV
5      do Enqueue(queue, v)
6    while queue ≠ ∅
7      do cur_v ← Dequeue(queue)
8         Add(visited, cur_v)
9         ▷ Проход по всем вершинам, смежными с вершиной cur_v
10        for each u in Adjacent(DB, cur_v)
11          do if u ∉ visited
12             then Enqueue(queue, u)
13   return visited
14
```

Рисунок 8 – Алгоритм поиска взаимосвязанных данных в метаграфе базы данных

Говоря о представленном выше описании, отметим следующие моменты:

– алгоритм принимает на вход два параметра: метаграф базы данных (может быть симметричным) *DB*, и множество начальных вершин *SV*, с которых начинается поиск,
– операция *Enqueue(Q, x)* предусматривает добавление элемента *x* в конец очереди *Q*,
– операция *Dequeue(Q)* удаляет элемент из начала очереди *Q*,
– операция *Add(S, x)* добавляет элемент *x* в множество *S*,
– функция *Adjacent(MG, x)* возвращает множество вершин, которые являются смежными с вершиной *x* в метаграфе *MG*.

2.2.4 Собственный метаграф

При применении алгоритма, представленного на рисунке 8, могут возникнуть определённые сложности:

– алгоритм имеет тенденцию возвращать избыточные данные, особенно в случае симметричного входного метаграфа. Рассмотрим ситуацию, когда тестировщику необходимы данные только по классу *Class 1A* из таблицы *classes*, включая всех учащихся, представленных в таблице *students*, взятых из базы данных, изображённой на рисунке 6. Но в случае симметричного метаграфа алгоритм, помимо вершин, соответствующих классам и учащимся, может вернуть также вершины, представляющие данные из других таблиц, таких как *lessons*, *subjects* и *teachers*;
– метарёбра и рёбра в метаграфе базы данных, формируются на основании внешних ключей. Тем не менее, возможно отсутствие внешнего ключа, даже когда таблицы имеют логическую связь. Это может быть следствием, например, денормализации данных [25].

Напрямую изменять метаграф, представляющий базу данных, нецелесообразно, так как это требует изменений в её структуре. Вместо этого возможно создание **собственного метаграфа** на основе исходного, с изменением множеств метарёбер и рёбер.

Но такой собственный метаграф не будет являться прямым отображением базы данных и также будет храниться в оперативной памяти компьютера. Это означает два важных аспекта:

– полное копирование множеств вершин и рёбер из метаграфа базы данных не будет производиться, поскольку это может привести к

значительному расходу памяти в случае большой базы данных. Вместо этого, множества вершин и рёбер собственного метаграфа будут формироваться динамически, в процессе выполнения алгоритма, с обращением к метаграфу базы данных. При этом множества метавершин и метарёбер будут скопированы без изменений, так как количество таблиц в базе данных обычно невелико по сравнению с количеством данных в этих таблицах;

– в собственном метаграфе будет иной набор атрибутов для каждой вершины. Этот аспект будет рассмотрен более подробно чуть позже.

2.2.5 Правила метаграфа

Введём понятие правила метаграфа, которое будет отвечать за изменение множеств метарёбер и рёбер. Соответственно, правила метаграфа можно разделить на два типа: те, которые модифицируют множество метарёбер, и те, которые изменяют множество рёбер.

Определим **правило метаграфа** следующим образом:

Пусть $r = \langle p, f \rangle$ представляет собой правило метаграфа, где p — предикат, а f — функция, ответственная за модификацию множества метаграфа.

Предикат p представляет собой условие, формализованное в виде булевой функции, которая получает в качестве входного параметра метаребро (или ребро) и возвращает значение *True* или *False*. Формально это можно представить так:

$p : e => \{True, False\}$, где $e \in ME$ (или $e \in E$), и ME (или E) обозначает множество метарёбер (или рёбер).

Функция f отвечает за изменение множества метарёбер (или рёбер). Она принимает на вход множество всех метавершин (или вершин) в метаграфе, возвращая новое множество метавершин (или вершин). Формально это выглядит следующим образом:

$f : E => E'$, где E – множество метарёбер (или рёбер) метаграфа, E' – обновлённое множество метарёбер (или рёбер).

Таким образом, правило метаграфа обеспечивает возможность изменения множества метарёбер (или рёбер), при выполнении определённого условия. Примером такого правила может служить следующее: удалить все рёбра, исходящие из вершины v, если эта вершина соответствует данным из

таблицы *classes*, где значение поля *class_id* равно 1.

Это определение предоставляет формальную структуру для описания правил, которые могут использоваться для целенаправленного изменения структуры метаграфа в зависимости от предикатов.

2.2.6 Описание алгоритма с собственным метаграфом и правилами метаграфа

Опишем алгоритм, который принимает на вход метаграф базы данных, множество начальных вершин, а также два множества правил метаграфа: одно для метарёбер, другое для рёбер.

На выходе алгоритм возвращает множество вершин, связанных с начальными вершинами, включая сами начальные вершины. В процессе работы алгоритм применяет правила метаграфа. Алгоритм представлен на рисунке 9.

```
1  Metagraph_Traversal_With_Rules(DB, SV, RME, RE)
2    ▷ Инициализация собственного метаграфа MG: копирование
       метавершин и метарёбер из метаграфа базы данных
3    MG ← <∅, ∅, ∅, ∅>
4    MG.MV ← Copy(DB.MV)
5    MV.ME ← Copy(DB.ME)
6    ▷ Применение правил RME к метарёбрам
7    MG.ME ← Apply_Rules(MG.ME, RME)
8    queue ← ∅
9    visited ← ∅
10   for each v in SV
11     do Enqueue(queue, v)
12   while queue ≠ ∅
13     do cur_v ← Dequeue(queue)
14        Add(visited, cur_v)
15        ▷ Дополнение метаграфа MG текущей вершиной
16        Add(MG.V, cur_v)
17        ▷ Получение инцидентных к вершине cur_v рёбер
18        incident_edges ← Incident(DB, cur_v)
19        ▷ Применение правил RE к полученным рёбрам
20        new_incident_edges ← Apply_Rules(incident_edges, RE)
21        ▷ Дополнение метаграфа MG новым множеством рёбер
22        Extend(MG.E, new_incedent_edges)
23        ▷ Проход по всем вершинам, смежными с вершиной cur_v,
       обновлённого метаграфа MG
24        for each u in Adjacent(MG, cur_v)
25          do if u ∉ visited
26             then Enqueue(queue, u)
27   return visited
28
```

Рисунок 9 – Алгоритм поиска взаимосвязанных данных в метаграфе базы данных с использованием правил метаграфа

В отношении приведённого выше описания алгоритма отметим следующие аспекты:

– алгоритм принимает на вход четыре параметра: метаграф базы данных *DB* (возможно симметричный), множество начальных

вершин *SV*, которые служат отправной точкой поиска, и два множества правил метаграфа: для метарёбер и для рёбер,

- функция *Copy(S)* возвращает копию множества S,
- функция *Incident(MG, x)* возвращает множество инцидентных вершине x рёбер, которые находятся в метаграфе MG,
- операция *Extend(S, V)* добавляет в множество S все элементы множества V,
- функция *Apply_Rules(S, R)* применяет множество правил R к элементам множества S, формируя новое множество.

Функция *Apply_Rules* изображена на рисунке 10.

```
1 Apply_Rules(S, R)
2   A ← Copy(S)
3   for each r in R
4     do for each x in S
5       do if r.p(x) = True
6          then A ← r.f(A)
7   return A
8
```

Рисунок 10 – Алгоритм применения правил

Таким образом, алгоритм, изображённый на рисунке 9, позволяет применять правила метаграфа, обеспечивая тем самым гибкость в управлении обходом метаграфа.

2.2.7 Атрибуты вершин и устройство Postgresql

Ввиду того, что все элементы собственного метаграфа сохраняются непосредственно в оперативной памяти компьютера, крайне важно минимизировать число и объём атрибутов вершин, одновременно обеспечивая возможность их сопоставления с соответствующими данными. Для достижения данной цели ознакомимся с некоторыми аспектами внутреннего устройства PostgreSQL. Далее рассмотрим два различных подхода.

Первый подход предполагает сохранение в качестве атрибутов вершины пары: наименование таблицы, в которой хранятся данные, и значение первичного ключа [26]. Подобная пара обеспечивает уникальную

идентификацию данных в пределах одной схемы базы данных [27]. Для достижения уникальности в рамках всей базы данных необходимо дополнительно учитывать наименование схемы. Следует отметить, что первичный ключ может быть составным [28], и в таком случае потребуется сохранять несколько значений, что увеличивает объём атрибутов. Более того, может отсутствовать сам первичный ключ; в таких обстоятельствах, если уникальность записи всё же гарантируется, нужно сохранять полные значения всех полей для точной идентификации записи.

Второй подход заключается в использовани пар системных полей *tableoid* и *ctid*. Поле *tableoid* указывает на уникальный идентификатор таблицы в пределах всей базы данных [29], а поле *ctid* определяет физическое расположение записи в таблице [30], занимая при этом всего 6 байт памяти [31]. Но существует несколько нюансов:

- для осуществления запроса все равно потребуется определить название таблицы. Это можно реализовать посредством объединения (*JOIN*) с системной таблицей *pg_class* [32]. Для избежания дополнительных операций при каждом запросе, возможно создание хеш-таблицы перед запуском алгоритма, где ключами выступают *tableoid*, а значениями — названия таблиц;
- неизменность *ctid* для записи гарантируется лишь в пределах одной транзакции. Весь алгоритм перемещения данных выполняется в рамках единой транзакции. Но в случае её прерывания из-за сбоя, после восстановления системы нет никакой гарантии, что сохранённые *ctid* в состоянии алгоритма, представляющее собой собственный метаграф, будут соответствовать прежним данным.

Таким образом, для выбора атрибутов вершины собственного метаграфа возможно использование одного из двух подходов: первый потенциально требует больше памяти, в то время как второй не обеспечивает гарантии корректного восстановления данных после сбоя.

2.3 Генерация данных

Рассмотрим алгоритм генерации данных и возможности настройки данного процесса пользователем.

Для функционирования алгоритма требуется схема базы данных, на основании которой будет производиться генерация данных. Пользователь

имеет возможность предоставить готовую схему или данные для подключения к базе данных, из которой будет извлечена соответствующая схема.

Метаданные должны содержать следующую информацию.

– множество допустимых значений для каждого поля данных. Если для определённых полей множество допустимых значений не указано явно, оно формируется на основе схемы базы данных, учитывая тип поля и его ограничения;
– объём генерируемых данных;
– названия таблиц, с которых начинается процесс генерации взаимосвязанных данных, и правила метаграфа.

Алгоритм генерации данных схож с алгоритмом 9, но проход будет осуществляться по метавершинам и метарёбрам.

Входными данными для такого алгоритма являются:

– метаграф без множества вершин и рёбер, отражающий схему базы данных,
– правила метаграфа, действующие исключительно на множество метарёбер, так как рёбра в алгоритме не используются,
– множество метавершин, для каждой из которых запускается алгоритм прохода по метарёбрам,
– объём генерируемых данных, который в простейшем варианте выражается в виде хеш-таблицы, где ключ — это метавершина, а значением является количество данных, подлежащих генерации,
– допустимые значения для генерации, представленные в виде хеш-таблицы, где ключом выступает атрибут метавершины, соответствующий полю таблицы, а значением — множество допустимых значений для данного поля.

Алгоритм на выходе предоставит множество вершин, атрибуты которых будут содержать сгенерированные данные. Алгоритм генерации взаимосвязанных данных представлен на рисунке 11.

```
1  Data_Generator(MG, RME, SMV, GC, AV)
2    MG.ME ← Apply_Rules(MG.ME, RME)
3    vertexes ← ∅
4    for each mv in SMV
5      do queue ← {mv}
6         visited ← ∅
7         while queue ≠ ∅
8           do cur_mv ← Dequeue(queue)
9              Add(visited, cur_mv)
10             Extend(vertexes, Gen(cur_mv, GC, AV))
11             for each mu in Adjacent(MG, cur_mv)
12               do if mu ∉ visited
13                  then Enqueue(queue, mu)
14   return vertexes
15
```

Рисунок 11 – Алгоритм генерации взаимосвязанных данных

В данном алгоритме:

- принимаются пять аргументов: метаграф *MG* (возможно симметричный) с множеством метавершин и метарёбер, правила метаграфа для метарёбер, множество начальных метавершин *SMV*, объём генерируемых данных *GC*, допустимые значения для генерации *AV*,
- функция *Gen(MV, GC, AV)* генерирует вершины, для которых данные соответствуют таблице, определяемой метавершиной MV, а *GC* и *AV* обозначают объём данных и допустимые значения для их генерации соответственно.

2.4 Разработка языка

Как упоминалась ранее, метаданные включают набор правил, описывающих способы переноса, генерации или анонимизации данных. Способ описания метаданных должен быть простым, понятным и гибким. В этом разделе рассматривается создание предметно-ориентированного языка программирования [33], предназначенного для описания метаданных.

2.4.1 Синтаксис и семантика языка

Разрабатываемый язык будет носить декларативный характер и использовать ряд SQL-операторов [34]. Опишем синтаксис и семантику языка [35].

Синтаксис будет представлен в расширенной форме Бэкуса-Наура [36]. Предположим, что для оператора *WHERE* уже определен нетерминальный символ *where_statement*, а такие символы, как *integer* для обозначения целого числа, *table* для идентификации названия таблиц, *field* для обозначения названия поля, и *function_name* для обозначения названия функции, уже введены. К тому же, все конструкции должны оканчиваться на символ ”;”.

Также будет указан синтаксис вызова функции *function_call*. Его определение показано на рисунке 12.

```
1 function_call = function_name, ["(", argument, {",", argument
    }, ")"];
2
```

Рисунок 12 – function_call

Синтаксис определения функции в рамках данной работы не рассматривается.

2.4.1.1 GRAPH SOURCE

С помощью конструкции, изображённой на рисунке 13, определяется стартовая вершина, данные которой находятся в таблице *table*. Если присутствует конструкция *where_statement*, применяется соответствующее фильтрование данных.

```
1 graph_source = "GRAPH SOURCE", " ", table, [" ",
    where_statement];
2
```

Рисунок 13 – GRAPH SOURCE

2.4.1.2 INCLUDE EDGE и EXCLUDE EDGE

Конструкции, изображённые на рисунке 14, позволяют включать или

исключать метаребро между двумя метавершинами, ассоциированными с указанными таблицами. Названия полей входят в атрибуты метаребра и означают, что ребро существует (или не существует), если для указанных полей значения совпадают.

```
1 include_edge = "INCLUDE EDGE", " ", table, ".", field, " ",
    table, ".", field;
2 exclude_edge = "EXCLUDE EDGE", " ", table, ".", field, " ",
    table, ".", field;
3
```

Рисунок 14 – INCLUDE EDGE и EXCLUDE EDGE

2.4.1.3 NO ENTER

Конструкция, изображённая на рисунке 15, позволяет удалить метарёбра, входящие в метавершину, соответствующую таблице *table*. В случае если задана конструкция *where_statement*, удаляются входящие в вершину рёбра, данные которых соответствуют фильтру.

```
1 no_enter = "NO ENTER", " ", table, [" ", where_statement];
2
```

Рисунок 15 – NO ENTER

2.4.1.4 NO EXIT

Посредством конструкции, изображённой на рисунке 16, удаляются метарёбра, выходящие из метавершины, ассоциированной с таблицей *table*. Если конструкция *where_statement* указана, это приводит к удалению выходящих из вершины рёбер, данные которых проходят фильтрацию.

```
1 no_exit = "NO EXIT", " ", table, [" ", where_statement];
2
```

Рисунок 16 – NO EXIT

2.4.1.5 LIMIT VISITS

Конструкция, изображённая на рисунке 17, позволяет ограничить число

посещений вершин, находящихся в метавершине, соответствующей таблице *table*. Параметр *integer* определяет максимальное количество посещений.

```
limit_visits = "LIMIT VISITS", " ", integer, " ", "FOR", " ",
    table;

```

Рисунок 17 – LIMIT VISITS

2.4.1.6 LIMIT DISTANCE

Конструкция, изображённая на рисунке 18, позволяет ограничить длину пути по метарёбрам. В частности, она способствует удалению всех метарёбер (и соответствующих рёбер), находящихся на расстоянии, превышающем *integer* метавершин от метавершины, связанной с таблицей *table*.

```
limit_distance = "LIMIT DISTANCE", " ", integer, " ", "FOR",
   " ", table;

```

Рисунок 18 – LIMIT DISTANCE

2.4.1.7 TRANSFORMER

Конструкция, изображённая на рисунке 19, позволяет установить трансформер для определённых полей, при этом он задаётся функцией. Поля таблиц, к которым нужно применить трансформер при переносе данных, указываются после ключевого слова *”FOR”*.

```
transformer = "TRANSFORMER", " ", function_call, " ", "FOR",
   " ", table, ".", field, {",", table, ".", field};

```

Рисунок 19 – TRANSFORMER

2.4.1.8 SET GENERATION VALUES

Конструкция, изображённая на рисунке 20, служит для определения множества, диапазона или функции, которые будут задавать значения для

генерации соответствующих полей. Здесь символ *value* представляет собой произвольную последовательность символов, применимую в качестве значения для поля.

```
1 set_of_values = "{", value, {",", value}, "}";
2 range_of_values = "[", integer, ":", integer, "]";
3 set_generation_values = "SET GENERATION VALUES", " ",
     set_of_values|range_of_values|function_call, " ", "FOR", "
      ", table, ".", field, {",", table, ".", field};
4
```

Рисунок 20 – SET GENERATION VALUES

2.4.1.9 SET GENERATION AMOUNT

С помощью конструкции, изображённой на рисунке 21, можно задать количество генерируемых данных для каждой указанной таблицы.

```
1 set_generation_amount = "SET GENERATION AMOUNT", " ", table,
     "=", integer, {", ", table, "=", integer};
2
```

Рисунок 21 – SET GENERATION AMOUNT

2.4.2 Примеры

Примеры рассматриваются на основе базы данных, изображённой на рисунке 6.

Метаданные, приведённые на рисунке 22, позволяют перенести все данные, связанные с классом, идентифицируемым как *class_id=1*, при этом исключая перенос таблиц *teachers*. Кроме того, данные учащихся будут анонимизированы: генерируются случайные имя и фамилия и устанавливается фиксированная дата рождения.

```
1 GRAPH SOURCE classes WHERE class_id=1;
2
3 NO ENTER teachers;
4
5 TRANSFORMER random_first_name FOR students.first_name;
6 TRANSFORMER random_last_name FOR students.last_name;
7 TRANSFORMER set_birth_data("2000-01-01") FOR students.
    birth_date;
8
```

Рисунок 22 – Пример описания метаданных 1

Рисунок 23 включает метаданные, описывающие перенос таблиц *lessons* и *subjects* со всеми данными и двух случайных учителей из таблицы *teachers*, для которых имя и фамилия будут заданы как *”Anon”*.

```
1 GRAPH SOURCE lessons;
2
3 EXCLUDE EDGE lessons.class_id classes.class_id;
4 LIMIT VISITS 2 FOR teachers;
5
6 TRANSFORMER set("Anon") FOR teachers.first_name, teachers.
    last_name;
7
```

Рисунок 23 – Пример описания метаданных 2

Рисунок 24 демонстрирует метаданные, описывающие процесс генерации данных.

```
GRAPH SOURCE lessons;

LIMIT DISTANCE 1 FOR lessons;

SET GENERATION VALUES gen_date("2020-01-01", "2025-01-01")
   FOR lessons.date;
SET GENERATION VALUES {"Class 1A", "Class 1B", "Class 1C"}
   FOR classes.name;
SET GENERATION VALUES gen_random_subject FOR subjects.name;

SET GENERATION AMOUNT lessons=10, classes=5, subjects=10;

```

Рисунок 24 – Пример описания метаданных 3

2.4.3 Грамматика языка и синтаксический анализатор

Рассмотрим разработку грамматики языка и реализацию синтаксического анализатора [37] с использованием инструмента ANTLR4 [38]. В качестве основы для построения грамматики была выбрана грамматика языка SQLite [39].

На рисунке 25 представлены основные конструкции в формате грамматики ANTLR4, которые ранее были описаны на основе расширенной формы Бэкуса-Наура.

```
1  graph_source_stmt: GRAPH_ SOURCE_ table_name (WHERE_ expr)?;
2  include_edge_stmt: INCLUDE_ EDGE_ table_name DOT column_name
     table_name DOT column_name;
3  exclude_edge_stmt: EXCLUDE_ EDGE_ table_name DOT column_name
     table_name DOT column_name;
4  no_enter_stmt: NO_ ENTER_ table_name (WHERE_ expr)?;
5  no_exit_stmt: NO_ EXIT_ table_name (WHERE_ expr)?;
6  limit_visits_stmt: LIMIT_ VISITS_ INTEGER_LITERAL FOR_
     table_name;
7  limit_distance_stmt: LIMIT_ DISTANCE_ INTEGER_LITERAL FOR_
     table_name;
8  transformer_stmt: TRANSFORMER_ function_call FOR_ table_name
     DOT column_name (COMMA table_name DOT column_name)*;
9  set_generation_values_stmt: SET_ GENERATION_ VALUES_ (
     set_of_values | range_of_values | function_call) FOR_
     table_name DOT column_name (COMMA table_name DOT
     column_name)*;
10 set_generation_amount_stmt: SET_ GENERATION_ AMOUNT_
     table_name ASSIGN INTEGER_LITERAL (COMMA table_name ASSIGN
      INTEGER_LITERAL)*;
11
```

Рисунок 25 – Описание основных конструкций в формате грамматики ANTLR4

На рисунке 27 представлен пример дерева синтаксического анализа, сгенерированного с помощью синтаксического анализатора, применительно к метаданным, изображённым на рисунке 26.

```
1  GRAPH SOURCE classes WHERE class_id=1;
2
3  NO ENTER teachers;
4
```

Рисунок 26 – Пример описания метаданных 4

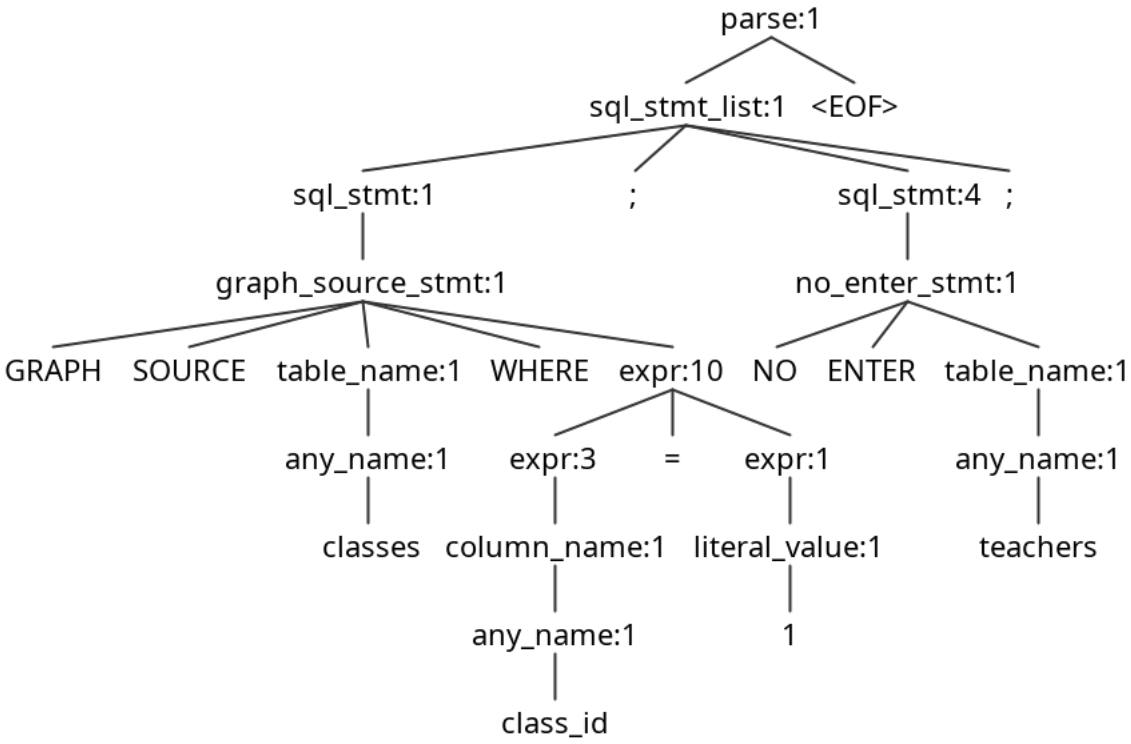

Рисунок 27 – Пример дерева синтаксического анализа

С использованием ANTLR4 на языке Python был сгенерирован синтаксический анализатор. Кроме того, была разработана программа, демонстрирующая возможности использования объектов ANTLR4.

Обход синтаксического дерева производится с использованием объекта ANTLR Слушатель (Listener), что позволяет реализовать соответствующую логику обработки для каждой конструкции. На рисунке 28 представлен код обработки конструкций *GRAPH SOURCE* и *NO ENTER*, который подразумевает простое отображение содержимого токенов. На рисунке 29 представлен вывод данной программы, соответствующий примеру метаданных, изображённым на рисунке 26.

```
1  from antlr4 import FileStream, CommonTokenStream,
     ParseTreeWalker
2  from RelatioLangLexer import RelatioLangLexer
3  from RelatioLangParser import RelatioLangParser
4  from RelatioLangParserListener import
     RelatioLangParserListener
5  from tools import get_original_token_text
6
7  class RelatioLangListener(RelatioLangParserListener):
8      def exitGraph_source_stmt(self, ctx:RelatioLangParser.
     Graph_source_stmtContext):
9          print("GRAPH SOURCE:")
10         print(f"\ttable is \"{ctx.table_name().getText()}\"")
11         if ctx.expr() is not None:
12             print(f"\twhere statement is \"{
     get_original_token_text(ctx.expr())}\"")
13
14     def exitNo_enter_stmt(self, ctx:RelatioLangParser.
     No_enter_stmtContext):
15         print("NO ENTER:")
16         print(f"\ttable is \"{ctx.table_name().getText()}\"")
17
18 def main():
19     filename = input()
20     input_stream = FileStream(filename)
21     lexer = RelatioLangLexer(input_stream)
22     stream = CommonTokenStream(lexer)
23     parser = RelatioLangParser(stream)
24     tree = parser.parse()
25     relatio_lang_listiner = RelatioLangListener()
26     walker = ParseTreeWalker()
27     walker.walk(relatio_lang_listiner, tree)
28
```

Рисунок 28 – Код с логикой обработки конструкций GRAPH SOURCE и NO ENTER

```
GRAPH SOURCE:
        table is "classes"
        where statement is "class_id=1"

NO ENTER:
        table is "teachers"

```

Рисунок 29 – Пример работы программы, обрабатывающей конструкции

Использование ANTLR4 для генерации синтаксического анализатора демонстрирует его эффективность в разборе и обработке грамматических конструкций. Такой подход не только упрощает процесс разработки грамматик, но и предоставляет гибкие возможности для расширения логики обработки, что позволяет интегрировать более сложные структуры и правила в дальнейшем.

3 Демонстрация программного продукта и анализ результатов

3.1 Обзор прототипа

В рамках дипломной работы был разработан прототип системы в виде инструмента с интерфейсом командной строки. Рассмотрим его функциональность.

Прототип обеспечивает перенос данных и их удаление из базы данных. Кроме того, данный инструмент позволяет осуществлять перенос и удаление схемы базы данных или выводить схему в формате диаграммы PlantUML [40].

В прототипе реализованы основные компоненты, указанные на рисунке 5, включая элементы Graph Walker и Data Writer. Процесс обхода данных основан на алгоритме, изображённом на рисунке 9. Для записи данных из одной базы в другую применяется механизм Foreign Data Wrapper.

Кроме того, в прототипе предусмотрены две версии взаимодействия с базой данных и между компонентами Graph Walker и Data Writer: синхронная и асинхронная. В случае синхронного взаимодействия компонент Graph Walker ожидает завершения записи данных компонентом Data Writer, в то время как при асинхронном взаимодействии компоненты функционируют независимо друг от друга.

Также в прототипе поддерживается функциональность следующих конструкции для описания метаданных: *GRAPH SOURCE*, *NO ENTER*, *NO EXIT*, *LIMIT DISTANCE*. Описываются метаданные в формате JSON [41].

3.2 Пример использования

Для иллюстрации использования будет использоваться база данных, изображённая на рисунке 6, которая будет выступать в роли источника. Метаданные для переноса представлены в формате JSON и изображены на рисунке 30.

```
{
  "source_rules": [
    {
      "table": "classes",
      "where": "class_id=1"
    }
  ],
  "traversal_rules": [
    {
      "type": "no_enter",
      "values": [
        {"table": "teachers"}
      ]
    }
  ]
}

```

Рисунок 30 – Метаданные в формате JSON

Предположим, имеется пустая целевая база данных, в которой отсутствует схема данных. Процесс переноса схемы базы данных, а также самих данных, можно осуществить с помощью двух команд, представленных на рисунке 31.

```
./prototype clone-schema --source-db postgresql://user:
  password@localhost:5432/source --target-db postgresql://
  user:password@localhost:5433/target

./prototype clone-data --source-db postgresql://user:
  password@localhost:5432/source --target-db postgresql://
  user:password@localhost:5433/target --rule-path rules.json

```

Рисунок 31 – Примеры запуска программы

После выполнения команд в целевой базе данных формируется копия схемы исходной базы, и соответствующие данные успешно переносятся.

На рисунке 32 отображены данные и их взаимосвязи в целевой базе данных. Можно отметить, что все данные, связанные с классом, имеющим идентификатор *class_id=1*, были успешно перенесены. При этом данные, содержащиеся в таблице *teachers*, не были включены в целевую базу данных.

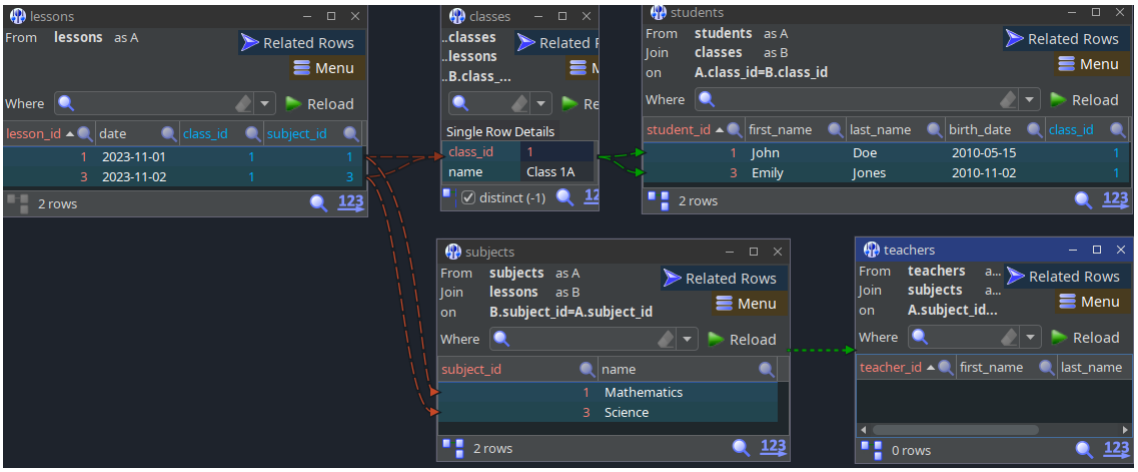

Рисунок 32 – Целевая база данных

3.3 Производительность

Проведём оценку производительности. В тестах производительности участвует асинхронная версия прототипа и программа, осуществляющая экспорт всех данных из исходной базы с помощью утилиты *pg_dump* и их вставку в целевую базу посредством инструмента *psql* – такую программу будет называть *pg_dump* + *psql*.

Для тестов была создана база данных, схема которой представлена на рисунке 6. Были сгенерированы данные для таблиц *students* и *classes* и помещены в базу данных. Общий объём данных составил около **4,5 гигабайтов**, а количество данных составило **44739072 записей**.

Допустим, есть пользователь, который должен проверить некоторую функциональность. Для проверки этой функциональности ему нужно перенести конкретные данные со всеми связными данными из созданной нами (исходной) базы в пустую (целевую) базу. При этом лишние данные не влияют на проверку. Пусть известно точное количество данных, необходимое пользователю, которого будет достаточно для проверки заданной функциональности. Такое количество данных будем называть *минимальным количеством записей*.

Было проведено 6 тестов. Каждый тест различался минимальным

количеством записей. Для каждого из тестов были написаны соответствующие метаданные. Тестирование проводилось на компьютере с системой *6.12.12-2-MANJARO*, процессором *AMD Ryzen 7 7700* и *32 GiB* оперативной памяти.

На рисунке 33 и в таблице 1 продемонстрировано время работы программ в зависимости от минимального количества записей. Очевидно, что время работы программы *pg_dump + psql* не зависит от минимального количества записей: чтобы сохранить реляционную целостность данных, программа *pg_dump + psql* переносит все данные. Прототип же умеет переносить взаимосвязанные данные, поэтому он переносит в точности минимальное количество записей, и, соответственно, время работы прототипа зависит от этого параметра.

Таблица 1 – Сравнение производительности

Минимальное количество записей	Время работы прототипа (мс)	Время работы pg_dump + psql (мс)
512	18970	395055
2048	24285	399688
8192	35052	393125
32768	67705	390495
131072	157820	396924
524288	411973	404070

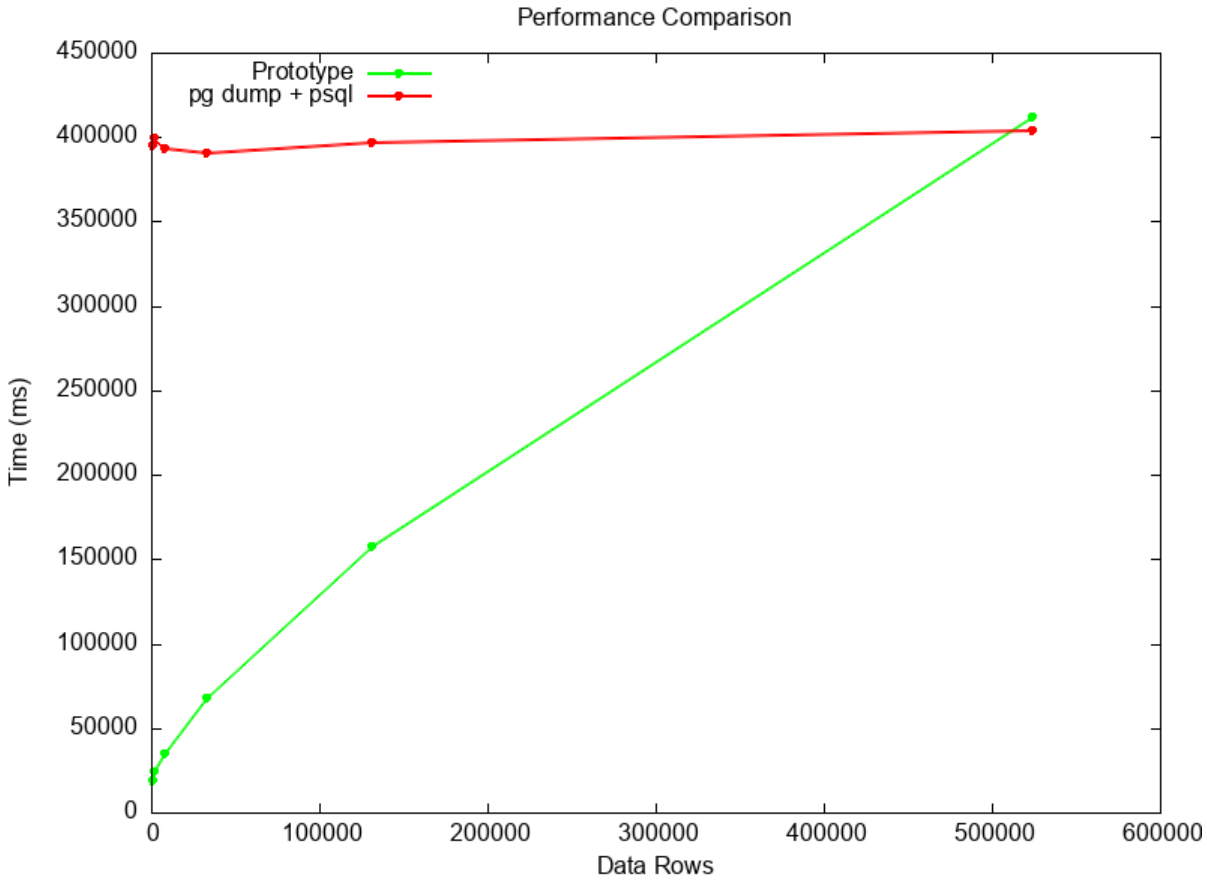

Рисунок 33 – График производительности

По проведённым тестам производительности можно сделать вывод: при переносе взаимосвязанных данных в объёме менее полумиллиона записей для заданной базы данных разработанный прототип демонстрирует более высокую скорость работы по сравнению с программой *pg_dump + psql*.

3.4 Анализ результатов

Проведём анализ жизнеспособности системы на основе спроектированной системы, разработанных алгоритма, языка и прототипа, а также результатов производительности.

Стоит отметить, что готового решения, удовлетворяющего всем описанным ранее требованиям, в виде единного программного продукта найдено не было. Несмотря на это, задачи переноса взаимосвязанных данных, генерации данных и анонимизации данных можно решить с помощью отдельных готовых инструментов. Например, с помощью инструмента Jailer [42] можно получить взаимосвязанные данные. Тем не менее, он обладает только графическим интерфейсом, что усложняет его интеграцию с системами

автоматизации. Кроме того, непосредственное подключение к производственной базе конечным пользователем представляет собой значительную угрозу безопасности данных, требующую устранения. Для анонимизации данных можно использовать инструмент Triki [43]. Но его нельзя использовать в процессе переноса данных: сначала данные нужно перенести, а только потом анонимизировать. В связи с этим, нужно обеспечить непрерывность процесса от переноса до анонимизации данных, чтобы пользователи целевой базы не смогли получить доступ к чувствительным данным. На основании изложенного можно заключить, что сборка рассматриваемой системы из готовых компонентов возможна, но потребует значительных доработок, возможно, соизмеримых по трудозатратам с разработкой новой системы.

Разработанным алгоритму выбора взаимосвязанных данных и языку можно сопоставить sql-запрос с применением CTE [44], который будет выбирать взаимосвязанные данные. Но такой запрос будет большим и он может требовать постоянной поддержки даже при незначительных изменениях в структуре базы. Особенность разработанных алгоритма и языка состоит в том, что человек, описывающий метаданные, должен хорошо знать структуру базы данных, либо пользоваться инструментами для анализа связей данных, такими как Jailer. В противном случае существует риск неэффективности процесса переноса данных: либо данных будет недостаточно, а значит, могут быть проблемы при тестировании на перенесённых данных; либо будет их избыток, что может значительно уменьшить производительность системы.

Результаты тестов производительности свидетельствуют о том, что при необходимости переноса взаимосвязанных данных в небольшом объёме (в рассматриваемом ранее примере этот объём составил около одного процента от всего объёма данных в БД) разработанный прототип демонстрирует более высокую скорость работы по сравнению с аналогичным решением по полному переносу данных.

Из вышесказанного можно сделать вывод о жизнеспособности системы. Но эффективность её практического применения обусловливается пониманием функциональных возможностей и ограничений системы. Оптимальная область применения данной системы ограничивается обработкой взаимосвязанных данных небольшого объёма.

ЗАКЛЮЧЕНИЕ

В ходе дипломной работы была спроектирована система переноса и генерации взаимосвязанных данных. Были детально описаны алгоритмы выбора и генерации взаимосвязанных данных, разработан специфальный язык для описания метаданных, а также создан прототип системы. Дополнительно проведено тестирование производительности прототипа и осуществлён анализ жизнеспособности системы.

Несмотря на то, что проектирование и разработка проводились при тестировании образовательной платформы и основное тестирование прототипа осуществлялось с использованием одной базы данных, предлагаемая система обладает универсальностью и может быть интегрирована в другие платформы и применена в различных областях. Важным аспектом архитектуры системы является то, что функции, характерные для работы с PostgreSQL, инкапсулированы в одном компоненте. Это способствует легкой адаптации системы к другим реляционным СУБД.

В процессе разработки и тестирования прототипа были выявлены места, требующих улучшения:

- первая проблема заключается в возникновении сложностей в понимании структуры базы данных и алгоритма обхода, что делает трудоёмким написание метаданных, которые бы удовлетворяли потребности пользователя. Возможным решением этой проблемы может стать создание инструмента для анализа структуры базы данных, аналогичного Jailer, который бы поддерживал описанный алгоритм обхода и правила метаграфа. Предполагаемый инструмент обеспечит пользователю возможность эффективной визуализации структуры базы данных и упростит процесс описания правил метаграфа;
- вторая проблема заключается в низкой производительности программы, особенно при больших объёмах данных. Есть гипотеза, что низкая производительность связана с частыми сетевыми запросами в базы данных. В качестве потенциального решения данной проблемы предлагается модификация взаимодействия с базой данных путём замены множественных мелких запросов на более редкие, но масштабные обращения к данным.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. TADVISER. Онлайн-образование (рынок России). — 05.02.2025. — URL: https://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:%D0%9E%D0%BD%D0%BB%D0%B0%D0%B9%D0%BD-%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D1%80%D1%8B%D0%BD%D0%BE%D0%BA_%D0%A0%D0%BE%D1%81%D1%81%D0%B8%D0%B8) (дата обращения 08.05.2025).

2. Панаско Н. Детское онлайн образование в 2025 году: актуально для детей и родителей. — 24.09.2024. — URL: https://vkusnovk.ru/blog/detskoe-onlain-obrazovanie (дата обращения 08.05.2025).

3. PostgreSQL. PostgreSQL: The world’s most advanced open source database. — URL: https://www.postgresql.org/ (дата обращения 02.01.2025).

4. PostgreSQL. PostgreSQL: Documentation: 17: pg_dump. — URL: https://www.postgresql.org/docs/current/app-pgdump.html (дата обращения 02.01.2025).

5. Rouhaud J. PostgreSQL: pg_anonymize, a new extension for simple and transparent data anonymization. — 09.03.2023. — URL: https://www.postgresql.org/about/news/pg_anonymize-a-new-extension-for-simple-and-transparent-data-anonymization-2606/ (дата обращения 02.01.2025).

6. lexone. Тестирование программного обеспечения. — 05.06.2024. — URL: https://www.lexone.ru/background-intel/software-testing.html (дата обращения 08.05.2025).

7. Блэк Р. Ключевые процессы тестирования. — Лори, 2025. — ISBN 978-5-85582-392-9.

8. Царев Ю. В., Сильянова Е. Ф., Кисельников С. А. ОСОБЕННОСТИ РУЧНОГО И АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ // Вестник науки. — 2021. — URL: https://cyberleninka.ru/article/n/osobennosti-ruchnogo-i-avtomatizirovannogo-testirovaniya-programmnogo-obespecheniya (дата обращения 09.05.2025).

9. ИНТУИТ Н. НОУ ИНТУИТ | Архитектурное проектирование программного обеспечения. Лекция 3: Разработка архитектуры программного обеспечения. Аналитический синтез информации. — URL:

https://intuit.ru/studies/courses/3509/751/lecture/29036?page=3 (дата обращения 09.05.2025).

10. Ричардс М., Форд Н. Фундаментальный подход к программной архитектуре: паттерны, свойства, проверенные методы. — Питер, 2023. — ISBN 978-5-4461-1842-7.

11. Model C. Home | C4 model. — URL: https : / / c4model . com/ (дата обращения 29.03.2025).

12. Озерицкий А. Logbroker: сбор и поставка больших объемов данных в Яндексе / Хабр. — 09.10.2014. — URL: https://habr.com/ru/companies/yandex/articles/239823/ (дата обращения 05.04.2025).

13. MrPizzly. OAuth 2.0 / Хабр. — URL: https://habr.com/ru/companies/beget/articles/886874/ (дата обращения 09.05.2025).

14. PostgreSQL. PostgreSQL: Documentation: 17: F.36. postgres_fdw — access data stored in external PostgreSQL servers. — URL: https://www.postgresql.org/docs/current/postgres-fdw.html (дата обращения 30.03.2025).

15. Семме О. Механизм доставки сообщений в Kafka | ADS Arenadata Docs Guide. — URL: https://docs.arenadata.io/ru/ADStreaming/current/concept/architecture/kafka/delivery_guarantees.html (дата обращения 05.04.2025).

16. Atłasik K. Microservices 101: Transactional Outbox and Inbox. — 03.06.2022. — URL: https : / / softwaremill . com / microservices - 101/ (дата обращения 05.04.2025).

17. Basu A., Robert W. B. Metagraphs and their applications. — New York: Springer, 2007.

18. Астанин С. В., Драгныш Н. В., Жуковская Н. К. Вложенные метаграфы как модели сложных объектов // Инженерный вестник Дона. — 2012. — URL: http://www.ivdon.ru/magazine/archive/n4p2y2012/1434 (дата обращения 09.05.2025).

19. Глоба С., Терновой М. Ю., Штогрина Е. С. Метаграфы как основа для представления и использования баз нечетких знаний // Открытые семантические технологии проектирования интеллектуальных систем = Open Semantic Technologies for Intelligent Systems (OSTIS-2015) : материалы V междунар. науч.-техн. конф. (Минск, 19-21 февраля 2015 года). — 2015. —

С. 237—240.

20. Самохвалов Э. Н., Ревунков Г. И., Гапанюк Ю. Е. Использование метаграфов для описания семантики и прагматики информационных систем // Вестник МГТУ им. Н. Э. Баумана. Сер. «Приборостроение». — 2015. — С. 83—99. — URL: https://vestnikprib.bmstu.ru/articles/673/673.pdf (дата обращения 09.05.2025).

21. Черненький В. М., Терехов В. И., Гапанюк Ю. Е. Представление сложных сетей на основе метаграфов // Нейроинформатика-2016. XVIII Всероссийская научно-техническая конференция. Сборник научных трудов. Ч. 1. М.: НИЯУ МИФИ, 2016. — 2016. — С. 173—178.

22. GeeksforGeeks. PostgreSQL – Foreign Key | GeeksforGeeks. — URL: https://www.geeksforgeeks.org/postgresql-foreign-key/ (дата обращения 12.04.2025).

23. Гришутин А., Алексеев С., Иванов М. Поиск в ширину - Алгоритмика. — URL: https://ru.algorithmica.org/cs/shortest-paths/bfs/ (дата обращения 16.04.2025).

24. Алгоритмы: построение и анализ / Т. Кормен [и др.]. — Вильям, 2011. — ISBN 978-5–8459–0857–5.

25. BillingMan. Зачем нужна денормализация баз данных, и когда ее использовать / Хабр. — 09.04.2016. — URL: https://habr.com/ru/companies/latera/articles/281262/ (дата обращения 15.04.2025).

26. PostgreSQL. PostgreSQL: Documentation: 17: 5.5. Constraints | DDL-CONSTRAINTS-PRIMARY-KEYS. — URL: https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-PRIMARY-KEYS (дата обращения 16.04.2025).

27. PostgreSQL. PostgreSQL: Documentation: 17: 5.10. Schemas. — URL: https://www.postgresql.org/docs/current/ddl-schemas.html (дата обращения 16.04.2025).

28. Malik T. S. Composite Primary Keys in PostgreSQL - CommandPrompt Inc. — 31.03.2023. — URL: https://www.commandprompt.com/education/composite-primary-keys-in-postgresql/ (дата обращения 16.04.2025).

29. PostgreSQL. PostgreSQL: Documentation: 17: 5.6. System Columns | DDL-SYSTEM-COLUMNS-TABLEOID. — URL: https://www.postgresql.org/docs/current/ddl-system-columns.html#DDL-SYSTEM-COLUMNS-TABLEOID (дата обращения 16.04.2025).

30. PostgreSQL. PostgreSQL: Documentation: 17: 5.6. System Columns | DDL-SYSTEM-COLUMNS-CTID. — URL: https://www.postgresql.org/docs/current/ddl-system-columns.html#DDL-SYSTEM-COLUMNS-CTID (дата обращения 16.04.2025).

31. PostgreSQL. PostgreSQL: Documentation: 17: 65.6. Database Page Layout. — URL: https://www.postgresql.org/docs/current/storage-page-layout.html#STORAGE-TUPLE-LAYOUT (дата обращения 16.04.2025).

32. PostgreSQL. PostgreSQL: Documentation: 17: 51.11. pg_class. — URL: https://www.postgresql.org/docs/current/catalog-pg-class.html (дата обращения 16.04.2025).

33. Languages D. S. How to create your own DSL from scratch. — URL: https://dsls.dev/article/How_to_create_your_own_DSL_from_scratch.html (дата обращения 23.04.2025).

34. ISO, IEC. ISO/IEC 9075-1:2023 - Information technology — Database languages SQL — Part 1: Framework (SQL/Framework). — URL: https://www.iso.org/standard/76583.html (дата обращения 24.04.2025).

35. Емельянов А. А. ЯЗЫКИ ПРОГРАММИРОВАНИЯ И ИХ СЕМАНТИКА В ТЕРМИНАХ ОТОБРАЖЕНИЙ СИНТАКСИСОВ // Научные проблемы водного транспорта. — 2015. — URL: https://cyberleninka.ru/article/n/yazyki-programmirovaniya-i-ih-semantika-v-terminah-otobrazheniy-sintaksisov (дата обращения 09.05.2025).

36. ISO, IEC. iso-14977.pdf. — URL: https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf (дата обращения 24.04.2025).

37. Погорелов Д. А., Таразанов А. М., Волкова Л. Л. От LR к Glr: обзор синтаксических анализаторов // Новые информационные технологии в автоматизированных системах. — 2017. — URL: https://cyberleninka.ru/article/n/ot-lr-k-glr-obzor-sintaksicheskih-analizatorov (дата обращения 09.05.2025).

38. ANTLR. ANTLR. — URL: https://www.antlr.org/ (дата обращения 01.05.2025).

39. antlr. grammars-v4/sql/sqlite at master · antlr/grammars-v4. — URL: https://github.com/antlr/grammars-v4/tree/master/sql/sqlite (дата обращения 01.05.2025).

40. PlantUML. Инструмент с открытым исходным кодом, использующий простые текстовые описания для рисования UML-диаграмм. — URL: https://plantuml.com/ru/ (дата обращения 27.04.2025).

41. JSON. JSON. — URL: https://json.org/json-ru.html (дата обращения 03.05.2025).

42. Jailer. Open Jail - The Jailer Project Web Site. — URL: https://wisser.github.io/Jailer/ (дата обращения 04.05.2025).

43. josacar. josacar/triki: Mysql, PostgreSQL and SQL dump obfuscator aka anonimizer. — URL: https://github.com/josacar/triki (дата обращения 04.05.2025).

44. PostgreSQL. PostgreSQL: Documentation: 9.6: WITH Queries (Common Table Expressions). — URL: https://www.postgresql.org/docs/9.6/queries-with.html (дата обращения 04.05.2025).

ПРИЛОЖЕНИЕ А
Исходный код грамматики на ANTLR4

На рисунке А.1 изображён QR-код со ссылкой на GitHub репозиторий с исходным кодом разработанной грамматики языка на ANTLR4.

Рисунок А.1 – QR-код на репозиторий с грамматикой