МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Дальневосточный федеральный университет»
(ДВФУ)

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

(ШКОЛА)

Департамент математического и компьютерного моделирования

ОТЧЕТ

о прохождении производственной практики

Педагогической практики

Направление подготовки
01.03.02 «Прикладная математика и информатика»

профиль «Системное программирование»

Выполнил студент гр.

Б9121-01.03.02сп1

Рютин Д. С. _____
(Ф.И.О.) *(подпись)*

Руководитель практики

Боровик А. И. _____
(должность, уч. звание) *(подпись)*

«21» апреля 2025 г.

Практика пройдена в срок
с «19» февраля 2025г.
по «22» апреля 2025г.

Отчет защищен:
с оценкой _____

Рег. № _____

«___» _____ 20___ г.

г. Владивосток

2025

Содержание

Введение

В современной разработке программного обеспечения, взаимодействия с базами данных является неотъемлемой частью большинства проектов. При во время работы над проектом разработчики сталкиваются с фундаментальной проблемой, известкой как ”объектно-реляционное несоответствие импеданса” (англ. object-relational impedance mismatch)

Объектно-реляционное несоответствие импеданса

Объектно-реляционное несоответствие импеданса – это набор трудностей, которые возникают при передаче данных между реляционными базами данных и моделями объектно-ориентированного программирования.

Различия концепций моделей

Объектно-ориентированный подход, с точки зрения математики, представляют собой направленные графы, где объект ссылается на другие дочерние. Реляционные модели основаны на кортежах в отношениях (таблицах) с использованием реляционной алгебры. Кортеж — это набор типизированных полей данных (строк). В реляционной модели связи обратимы, представляя собой ненаправленные графы.

Различия в объектно-ориентированных концепциях

Инкапсуляция: В ООП инкапсуляция скрывает внутреннее состояние объектов. Свойства объекта доступны только через реализованные интерфейсы.

Доступность: «Приватное» и «публичное» в реляционных системах зависит от потребностей. В ООП это абсолютно зависит от классов. Эта относительность и абсолютизм классификаций и характеристик противоречат друг

другу.

Интерфейсы, классы, наследование и полиморфизм: В ООП модели объекты должны реализовывать интерфейсы для предоставления доступа к своему внутреннему состоянию. В реляционной модели используются представления (views) для изменения перспективы и ограничений. Реляционная модель не имеет таких ООП концепций, как классы, наследование и полиморфизм.

Различия в типах данных

Реляционная модель запрещает ссылки, в то время как ООП модель активно их использует. Скалярные типы также часто различаются между моделями, что затрудняет отображение.

SQL поддерживает строки с максимальной длиной, обеспечивая большую производительность, и правила сортировки. В ООП библиотеках правила сортировки реализуются только через отдельные процедуры сортировки, а строки ограничены только объемом доступной памяти.

Манипулятивные различия

Реляционная модель использует стандартизированные операторы для работы с данными, в то время как ООП использует императивный подход для каждого класса и каждого случая. Любая декларативная поддержка ООП предназначена для списков и хеш-таблиц, в отличие от множеств в реляционной модели.

Транзакционные различия

В реляционной модели единицей работы является транзакция, которая превышает по размеру любые методы ООП классов. Транзакции могут включать произвольные комбинации операций с данными, тогда как в ООП модели есть только отдельные присваивания примитивным полям. ООП модель не

обеспечивает изоляцию и долговечность, так что атомарность и согласованность доступны только на уровне примитивов.

Различия в философии

- **Декларативный и императивный подходы** — Реляционная модель использует декларативный подход к данным, в то время как ООП использует императивное поведение.
- **Привязка к схеме** — Реляционная модель ограничивает строки их схемами сущностей. ООП наследование (древовидное или нет) аналогично.
- **Правила доступа** — в реляционных системах используются стандартизированные операторы, в то время как в ОО-классах есть отдельные методы. ОО-системы более выразительны, но реляционные системы отличаются математической логикой, целостностью и согласованностью дизайна.
- **Идентичность объекта** — Два изменяемых объекта с одинаковым состоянием различны. Реляционная модель игнорирует эту уникальность и должна искусственно создавать её с помощью кандидатных ключей, что является плохой практикой, если этот идентификатор не существует в реальном мире. Идентичность постоянна в реляционной модели, но может быть временной в ООП.
- **Нормализация** — ООП пренебрегает реляционной нормализацией.
- **Наследование схемы** — Реляционные схемы отвергают иерархическое наследование ООП. Реляционная модель принимает более мощную теорию множеств. Существует непопулярное недревовидное (не-Java) ООП, но оно сложнее, чем реляционная алгебра.

- **Структура и поведение** — ООП фокусируется на структуре (поддерживаемость, расширяемость, повторное использование, безопасность). Реляционная модель фокусируется на поведении (эффективность, адаптивность, отказоустойчивость, живучесть, логическая целостность и т.д.).

Итог

Использование реляционной базы данных для хранения объектно-ориентированн данных приводит к семантическому разрыву, который заставляет программистов постоянно преобразовывать данные между двумя разными формами представления. Эта необходимость не только снижает производительность разработки, но и создает дополнительные трудности, поскольку обе формы данных накладывают ограничения друг на друга.

Поскольку в реляционных базах данных информация хранится в виде набора таблиц с простыми данными, часто для хранения одного объекта необходимо использовать несколько таблиц, что требует применения операции JOIN для получения всей информации об объекте. Например, для хранения данных адресной книги могут использоваться как минимум две таблицы: люди и адреса, а возможно, и дополнительная таблица с телефонными номерами.

Традиционные СУБД обычно не оптимизированы для эффективной работы с объектно-ориентированными структурами через множество последовательных запросов. Последовательные запросы могут быть затратными и приводить к избыточным операциям чтения и записи. Один сложный запрос, охватывающий все связанные данные, обычно выполняется быстрее серии простых запросов.

ORM как решение проблемы несоответствия

Объектно-реляционное отображение (англ. Object-Relational Mapping, ORM) — это один из концептов работы с СУБД. ORM позволяет объеденить объектно-

ориентированный подход к программированию и реляционные базы данных.

Решение через ORM

ORM преодолевает разрыв между объектно-ориентированным программированием и реляционными базами данных следующим образом:

- **Автоматическое преобразование данных:** ORM создаёт соответствие между классами и таблицами, атрибутами объектов и полями таблиц, отношениями между объектами и связями между таблицами;
- **Абстрагирование от SQL:** Разработчики могут манипулировать данными, используя синтаксис и методы объектно-ориентированного программирования, без необходимости писать SQL-запросы;
- **Кэширование и оптимизация:** Многие ORM-системы включают механизмы кэширования и оптимизации запросов для повышения производительности.

С точки зрения программиста, ORM делает базу данных похожей на постоянное хранилище объектов. Разработчики могут просто создавать объекты и работать с ними как обычно, а фреймворк обеспечивает их сохранение в реляционной базе данных.

Преимущества и недостатки ORM

Преимущества:

- **Упрощает работу с базами данных.** Вам больше не нужно писать SQL-запросы — все действия с данными выполняются с использованием методов объектов;

- **Повышает производительность разработки.** Применяя готовые решения, разработчики могут сконцентрироваться на логике приложения, а не на низкоуровневых операциях с базой данных;
- **Уменьшение зависимости от конкретной базы данных.** При смене СУБД достаточно изменить конфигурацию ORM — это снижает риски и затраты на рефакторинг программного кода;
- **Автоматизация рутинных задач.** Многие из них, такие как создание индексов, миграции схем базы данных, автоматическое кэширование и управление транзакциями, реализуются встроенными механизмами ORM.

Недостатки:

- **Производительность.** Иногда применение ORM ведёт к снижению производительности — генерируемые запросы могут быть не такими оптимальными, как написанные вручную;
- **Ограниченная гибкость.** Некоторые специфические запросы или особенности конкретной БД могут оказаться недоступными через стандартный интерфейс ORM;
- **Зависимость от сторонних библиотек.** Из-за дополнительных библиотек и зависимостей возможны усложнения сборки и развертывания проекта;
- **Зависимость от конкретной реализации.** У различных ORM-решений есть свои особенности и ограничения, поэтому выбор конкретного инструмента требует тщательного анализа.

ORM в контексте C++

Разработка ORM для C++ представляет особые сложности из-за специфики языка:

- **Отсутствие встроенной рефлексии** в C++ усложняет автоматическое отображение классов на таблицы базы данных;
- **Строгая типизация** требует тщательного проектирования интерфейсов для обеспечения типобезопасности при работе с данными;
- **Отсутствие стандартного управления памятью** (сборки мусора, garbage collector) требует особого внимания к управлению жизненным циклом объектов;
- **Производительность** является критическим фактором для многих C++ проектов, что требует оптимизации генерируемого ORM SQL-кода.

Существующие ORM-решения для C++ часто используют метапрограммирование, генерацию кода или макросы для преодоления этих ограничений. Разработка эффективного ORM для C++ требует баланса между удобством использования и производительностью, с учетом специфических особенностей языка и типичных сценариев его применения.

Альтернативные решения проблемы несоответствия

Решения начинаются с осознания различий между логическими системами, что позволяет минимизировать или компенсировать несоответствие.

NoSQL: Объектно-реляционное несоответствие возникает только между ООП и системой управленя реляционной базой данных. Альтернативы, такие как NoSQL или базы данных XML, позволяют этого избежать.

Функционально-реляционное отображение: Термины функциональных языков программирования изоморфны реляционным запросам. Некоторые функциональные языки программирования реализуют функционально-реляционное отображение. Прямое соответствие между терминами и запросами позволяет избежать многих проблем объектно-реляционного отображения.

Минимизация в ОО: Минимизация объектно-ориентированного подхода (ОО) представляет собой методологию, направленную на преодоление объектно-реляционного несоответствия путем сознательного ограничения использования объектно-ориентированных концепций при проектировании взаимодействия с базами данных.

Данный метод базируется на следующих фундаментальных принципах:

- **Признание ограничений ОО-парадигмы:** Исследования показывают, что объектно-ориентированные базы данных (OODBMS) демонстрируют меньшую эффективность по сравнению с реляционными системами, поскольку объектно-ориентированный подход предоставляет недостаточно строгую структуру для проектирования схем данных;

- **Динамическое сопоставление:** В рамках этого подхода соответствие между объектами и реляционными структурами устанавливается во время выполнения программы, а не жестко фиксируется на этапе разработки;

- **Использование специализированных классов:** Внедрение классов кортежей и отношений в архитектуру приложения, что позволяет более естественно представлять реляционные данные в объектно-ориентированном контексте.

Таким образом, минимизация ОО представляет собой прагматический подход, который признает фундаментальные различия между объектно-ориентированной и реляционной парадигмами и предлагает компромиссное решение, сфокусированное на оптимальном представлении данных.

Основная часть

Выбор технологий для реализации ORM

Требования к современным ORM-решениям в C++

Современные ORM-решения для C++ должны эффективно преодолевать несколько ключевых проблем:

- **Отсутствие встроенной рефлексии** в C++, что усложняет автоматическое сопоставление классов с таблицами базы данных;
- **Необходимость типобезопасности** при преобразовании данных между языком программирования и базой данных;
- **Высокие требования к производительности** характерные для проектов на C++;
- **Минимизация рутинного кода** для определения схемы данных и операций с базой данных.

Подходы к решению проблемы отсутствия рефлексии

Рефлексия — это способность программы исследовать и модифицировать свою структуру и поведение во время выполнения. В языках со встроенной рефлексией (Java, C#, Python) разработчики могут легко получать информацию о классах, их полях и методах, что критически важно для ORM. C++ изначально не имеет встроенных механизмов рефлексии, что требует применения альтернативных подходов.

Макросы и препроцессор

Макросы в C++ позволяют автоматизировать генерацию кода, необходимого для описания метаданных классов:

- **Принцип работы:** Определение специальных макросов, которые разворачиваются в код для регистрации полей класса и их типов;
- **Примеры использования:**

```
class User {
    DECLARE_ENTITY(User)
    DEFINE_PROPERTY(int, id)
    DEFINE_PROPERTY(std::string, name)
    DEFINE_PROPERTY(std::string, email)
};
```

- **Преимущества:**
  - Простота реализации;
  - Не требует внешних инструментов;
  - Минимальные накладные расходы во время выполнения.
- **Недостатки:**
  - Ухудшает читаемость кода;
  - Затрудняет отладку (ошибки в макросах сложно диагностировать);
  - Ограниченная гибкость и выразительность;
  - Необходимость явного определения метаданных для каждого класса.
- **Примеры библиотек:** ODB, SOCI, SQLiteCpp используют макросы для отображения классов на таблицы БД.

Шаблонное метапрограммирование

Шаблоны C++ предоставляют мощный механизм для метапрограммирования времени компиляции:

- **Принцип работы:** Использование шаблонов и специализаций для создания типо-зависимой логики, которая генерирует метаданные о классах на этапе компиляции;
- **Примеры использования:**

```
template <typename T>
struct Schema;

template <>
struct Schema<User> {
    static constexpr auto metadata() {
        return make_metadata(
            field("id", &User::id),
            field("name", &User::name),
            field("email", &User::email)
        );
    }
};
```

- **Преимущества:**
  - Полная типобезопасность;
  - Ошибки выявляются на этапе компиляции;
  - Нет накладных расходов во время выполнения;
  - Более чистый синтаксис по сравнению с макросами.
- **Недостатки:**
  - Увеличивает время компиляции;
  - Усложняет сообщения об ошибках;
  - Требует специализации для каждого класса;
  - Ограничено возможностями шаблонов C++.
- **Примеры библиотек:** Boost.Hana, Boost.Fusion, magic_get используют шаблонное метапрограммирование для создания подобий рефлексии.

Кодогенерация на основе внешних схем

Данный подход использует внешние инструменты для генерации кода на основе схем или аннотаций:

- **Принцип работы:** Определение схемы данных во внешнем файле или с помощью специальных аннотаций, которые обрабатываются инструментом кодогенерации для создания необходимых классов и метаданных;
- **Примеры использования:**

```
// schema.proto
message User {
    required int32 id = 1;
    required string name = 2;
    required string email = 3;
}

// Запуск: protoc --cpp_out=. schema.proto
```

- **Преимущества:**
  - Чистый и читаемый код без макросов;
  - Единое определение схемы данных;
  - Возможность генерации кода для разных языков;
  - Разделение описания данных и логики их обработки;
  - Возможность расширения схемы через плагины кодогенерации.
- **Недостатки:**
  - Требует дополнительного шага в процессе сборки;
  - Дополнительные инструменты и зависимости;
  - Потенциальная сложность интеграции с IDE;
  - Ограничения выразительности определяемые форматом схемы.

- **Примеры технологий:** Protocol Buffers, FlatBuffers, Cap’n Proto, Apache Thrift.

Библиотеки статической рефлексии

Ряд библиотек предлагает решения для статической рефлексии в C++ без изменения стандарта языка:

- **Принцип работы:** Использование комбинации метапрограммирования, макросов и иногда предварительной обработки кода для создания возможностей рефлексии;
- **Примеры библиотек:**
  - **rttr (Run Time Type Reflection)** — предоставляет динамическую рефлексию для C++;
  - **refl-cpp** — легковесная, основанная только на заголовочных файлах библиотека для статической рефлексии;
  - **nameof** — позволяет получать имена типов, функций, переменных во время компиляции.
- **Преимущества:**
  - Более полная поддержка рефлексии по сравнению с ручными методами;
  - Часто имеют более чистый API;
  - Некоторые поддерживают динамические возможности рефлексии времени выполнения.
- **Недостатки:**
  - Дополнительные зависимости;
  - Возможно снижение производительности;

– Не являются стандартизированными решениями.

Будущее рефлексии в C++

Стоит отметить, что рефлексия активно обсуждается в комитете по стандартизации C++:

- **Предложения по стандартизации:** Существуют предложения по добавлению рефлексии в стандарт C++ (например, P2996, P1306, P3096);
- **Статическая рефлексия:** Предлагаемые решения фокусируются преимущественно на статической рефлексии времени компиляции;
- **Сроки:** Возможное включение некоторых возможностей рефлексии ожидается в C++26 или более поздних версиях.

Почему выбор пал на кодогенерацию

После анализа различных подходов к преодолению отсутствия рефлексии в C++, кодогенерация представляется наиболее оптимальным решением для реализации ORM по следующим причинам:

- **Чистота кода:** В отличие от макросов, кодогенерация позволяет разделить описание схемы данных и логику приложения, что значительно повышает читаемость и поддерживаемость кода.
- **Производительность:** По сравнению с библиотеками статической рефлексии, кодогенерация обеспечивает более высокую производительность за счёт отсутствия дополнительных слоёв абстракции во время выполнения.
- **Возможность расширения:** Кодогенерация предоставляет гибкий механизм для расширения функциональности через плагины и настройки генератора, что критически важно для ORM-специфичных задач.

- **Типобезопасность:** Генерируемый код проверяется компилятором, обеспечивая полную типобезопасность на уровне языка, что превосходит возможности макросов.

- **Кросс-платформенность:** Генераторы кода обычно поддерживают несколько языков программирования, что упрощает интеграцию с другими частями системы и возможное будущее расширение функциональности.

- **Единое представление данных:** Схемы данных, определённые для кодогенерации, могут использоваться не только для ORM, но и для сериализации, RPC и документирования, создавая единую точку истины.

В то время как шаблонное метапрограммирование предлагает элегантные решения для некоторых аспектов проблемы, оно значительно усложняет процесс компиляции и отладки. Библиотеки статической рефлексии, хотя и предоставляют богатый API, вводят дополнительные зависимости и часто имеют ограничения в производительности.

Кодогенерация предлагает оптимальный баланс между удобством разработки, производительностью и гибкостью, что делает её предпочтительным подходом для решения задач ORM в C++.

Сравнение кодогенераторов для решения задач ORM

Protocol Buffers

- **Производительность**:
  - Высокоэффективная бинарная сериализация
  - Быстрая десериализация с минимальными затратами памяти
  - Компактное бинарное представление данных

- **Форматы и преобразования**:

  - Встроенная поддержка преобразования в JSON и обратно
  - Компактный бинарный формат для эффективного хранения

- **RPC и сетевое взаимодействие**:
  - Тесная интеграция с высокопроизводительным gRPC фреймворком
  - Поддержка двунаправленного потокового взаимодействия

- **Поддержка C++**:
  - Генерация идиоматичного, эффективного C++ кода
  - Отличная интеграция с системами сборки (CMake, Bazel)

- **Расширяемость**:
  - Гибкая система пользовательских опций для метаданных
  - Расширяемость через плагины к компилятору protoc
  - Отличная поддержка версионирования схем данных

- **Экосистема**:
  - Активная поддержка со стороны Google
  - Обширная документация и сообщество
  - Проверенное временем решение с широким применением

FlatBuffers

- **Производительность**:
  - Zero-copy доступ к данным без полной десериализации
  - Более быстрое чтение отдельных полей
  - Возможность прямого редактирования сериализованных данных

- **Форматы и преобразования**:
  - Отсутствие встроенной поддержки преобразования в JSON
  - Больший размер сериализованных данных по сравнению с Protobuf
- **RPC и сетевое взаимодействие**:
  - Базовая поддержка RPC без развитой инфраструктуры gRPC
  - Отсутствие встроенной поддержки потокового взаимодействия
- **Поддержка C++**:
  - Хорошая поддержка C++, но менее идиоматичный API
  - Более сложная интеграция со сборочными системами
- **Расширяемость**:
  - Ограниченные возможности расширения для ORM-специфичных метаданных
  - Базовая поддержка версионирования схем
- **Экосистема**:
  - Меньшее сообщество и экосистема
  - Более ограниченная документация

Cap’n Proto

- **Производительность**:
  - Zero-copy доступ к данным без полной десериализации
  - Очень быстрое чтение отдельных полей
  - Эффективная работа с большими структурами данных

- **Форматы и преобразования**:
  - Отсутствие встроенной поддержки преобразования в JSON
  - Больший размер сериализованных данных по сравнению с Protobuf
- **RPC и сетевое взаимодействие**:
  - Собственный RPC-фреймворк с меньшей функциональностью чем gRPC
  - Ограниченная поддержка потоковой передачи данных
- **Поддержка C++**:
  - Хорошая поддержка C++, но менее идиоматичный API
  - Более сложная интеграция со сборочными системами
- **Расширяемость**:
  - Ограниченные возможности расширения для ORM-специфичных метаданных
  - Хорошая поддержка версионирования схем
- **Экосистема**:
  - Меньшее сообщество и экосистема
  - Более ограниченная документация

Apache Thrift

- **Производительность**:
  - Эффективная сериализация, но медленнее чем у Protocol Buffers
  - Нет поддержки zero-copy доступа к данным
  - Средний размер сериализованных данных

- **Форматы и преобразования**:
  - Поддержка нескольких форматов сериализации
  - Базовая поддержка преобразования в JSON
- **RPC и сетевое взаимодействие**:
  - Встроенный RPC-фреймворк с меньшей производительностью чем gRPC
  - Ограниченная поддержка потоковой передачи данных
- **Поддержка C++**:
  - Хорошая поддержка C++, но менее оптимизированный код
  - Более сложная интеграция со сборочными системами
- **Расширяемость**:
  - Ограниченные возможности расширения для ORM-специфичных метаданных
  - Ограниченная поддержка версионирования схем
- **Экосистема**:
  - Активное сообщество, но меньше чем у Protocol Buffers
  - Хорошая документация

JSON Schema

- **Производительность**:
  - Низкая эффективность сериализации/десериализации
  - Отсутствие компактного представления данных
  - Значительно больший размер данных

- **Форматы и преобразования**:
  - Нативная поддержка JSON
  - Отсутствие бинарного представления
- **RPC и сетевое взаимодействие**:
  - Отсутствие встроенного RPC-фреймворка
  - Необходимость использования сторонних решений (REST, GraphQL)
- **Поддержка C++**:
  - Ограниченная поддержка кодогенерации для C++
  - Сложная интеграция со сборочными системами
- **Расширяемость**:
  - Хорошие возможности для определения метаданных
  - Слабая поддержка версионирования схем
- **Экосистема**:
  - Широкое применение в веб-средах
  - Отсутствие единого стандарта кодогенерации

XML Schema (XSD)

- **Производительность**:
  - Очень низкая эффективность сериализации/десериализации
  - Очень большой размер сериализованных данных
  - Высокие затраты CPU и памяти при обработке
- **Форматы и преобразования**:

  - Нативная поддержка XML
  - Возможность преобразования в JSON через XSLT

- **RPC и сетевое взаимодействие**:
  - Отсутствие современного встроенного RPC-фреймворка
  - Устаревшие протоколы (SOAP)

- **Поддержка C++**:
  - Ограниченная поддержка кодогенерации для C++
  - Сложная интеграция со сборочными системами

- **Расширяемость**:
  - Богатая система аннотаций и расширений
  - Избыточная сложность для большинства задач

- **Экосистема**:
  - Устаревающие технологии с уменьшающимся сообществом
  - Применение преимущественно в старых корпоративных системах

Сравнительная таблица

Критерий	Protocol Buffers	FlatBuffers	Cap'n Proto	Apache Thrift	JSON Schema	XML Schema
Скорость сериализации	Высокая	Средняя	Высокая	Средняя	Низкая	Очень низкая
Скорость десериализации	Высокая	Очень высокая	Очень высокая	Средняя	Низкая	Очень низкая
Размер данных	Компактный	Средний	Средний	Средний	Большой	Очень большой
Поддержка JSON	Встроенная	Отсутствует	Отсутствует	Базовая	Нативная	Через конвертеры
RPC-фреймворк	gRPC	Базовый	Средний	Встроенный	Отсутствует	SOAP
Поддержка C++	Отличная	Хорошая	Хорошая	Хорошая	Ограниченная	Ограниченная
Расширяемость для ORM	Высокая	Средняя	Средняя	Средняя	Средняя	Высокая
Зрелость и сообщество	Очень высокая	Средняя	Средняя	Высокая	Высокая	Затухающая
Простота использования	Высокая	Средняя	Средняя	Средняя	Высокая	Низкая

Таблица 1: Сравнение кодогенераторов для ORM в C++

Преимущества Protocol Buffers для ORM в C++

- **Комплексная экосистема**:
  - Единая технология для описания данных, RPC и ORM
  - Возможность использования одних и тех же моделей для базы данных, API и межсервисного взаимодействия
- **Баланс производительности и удобства**:
  - Высокая эффективность без избыточного усложнения API
  - Компактное представление данных с сохранением читаемости кода
- **Универсальность форматов**:
  - Бинарный формат для высокопроизводительных сценариев
  - Встроенное преобразование в JSON для интеграции с веб-технологиями
- **Оптимальная поддержка для C++**:
  - Идиоматичный и эффективный генерируемый код
  - Отличная интеграция с типичными C++ системами сборки
- **Расширяемость для ORM**:
  - Гибкая система пользовательских опций для ORM-специфичных метаданных
  - Возможность расширения компилятора протоколов через плагины

Заключение

Protocol Buffers выбран как оптимальное решение для реализации ORM в C++ благодаря сочетанию высокой производительности, гибкости и развитой экосистеме. Интеграция с gRPC и поддержка JSON обеспечивают универ-

сальность для всех аспектов приложения. По сравнению с другими решениями, Protocol Buffers предлагает для меня наилучший баланс качеств.

Архитектура ORM

Инверсия зависимостей между ООП и реляционными моделями

Суть ORM заключается в инверсии зависимостей между объектно-ориентированн моделью и реляционной моделью данных. Это фундаментальный принцип, позволяющий разрешить объектно-реляционное несоответствие импеданса.

В объектно-ориентированном программировании данные представлены в виде сложных графовых структур объектов, обладающих:

- **Иерархиями наследования** — классы образуют древовидные или сетевые структуры;
- **Инкапсуляцией внутреннего состояния** — данные скрыты и доступны только через методы;
- **Полиморфизмом и динамическим поведением** — объекты могут менять поведение во время выполнения;
- **Разнообразными связями между объектами** — отношения 1:1, 1:N, M:N реализованные через ссылки;
- **Сложными типами данных** — композиция объектов внутри других объектов.

В реляционной модели данные организованы в таблицы с плоской структурой, связанные через ключи, с акцентом на:

- **Нормализацию данных** — разделение данных для устранения избыточности;

- **Ссылочную целостность** — обеспечение корректности связей через внешние ключи;
- **Атомарные типы данных** — хранение простых типов в ячейках таблиц;
- **Операции реляционной алгебры** — манипуляция данными через JOIN, SELECT, PROJECT и другие.

ORM инвертирует традиционный процесс разработки, позволяя разработчикам:

- Проектировать приложение, основываясь на объектной модели;
- Работать с объектами, как естественными сущностями вашего домена;
- Делегировать ORM трансформацию между объектным и реляционным представлениями.

Мое решение

Разрабатываемая ORM будет иметь модульную архитектуру из трех основных компонентов:

1. Ядро ORM

Этот компонент является центральным элементом системы и отвечает за:

- **Объектно-реляционное отображение:** автоматическое преобразование между объектами и таблицами;
- **Сессии и единицы работы:** отслеживание изменений в загруженных объектах;
- **Идентификационное отображение:** сохранение и восстановление идентичности объектов;
- **Ленивую загрузку:** отложенная загрузка связанных объектов;

- **Типобезопасные запросы:** DSL для построения запросов в объектно-ориентированном стиле.

Ядро будет использовать метаданные, сгенерированные из Protocol Buffers схем, для корректного отображения объектов на таблицы.

2. Master

Master служит интерфейсом между ядром ORM и базами данных:

- **Генерация SQL:** трансляция объектно-ориентированных запросов в оптимизированный SQL;
- **Управление соединениями:** пулы соединений и транзакций с БД;
- **Диалекты SQL:** адаптация запросов под особенности конкретных СУБД;
- **Преобразование данных:** маппинг между типами C++ и типами БД;
- **Обработка больших объемов данных:** пакетные операции и потоковая обработка.

Master абстрагирует ядро ORM от специфики конкретных СУБД и обеспечивает эффективное выполнение запросов.

3. Migrator

Migrator автоматизирует эволюцию схемы базы данных:

- **Создание и обновление схем:** автоматическая генерация DDL-скриптов;
- **Версионирование:** управление версиями схемы БД;
- **Дифференциальный анализ:** выявление изменений в схеме между версиями;
- **Безопасные миграции:** обеспечение сохранности данных при изменении схемы;

- **Интеграция с CI/CD:** автоматизация миграций в процессе развертывания.

Migrator обеспечивает синхронизацию между объектной моделью приложения и реляционной схемой БД.

Взаимодействие компонентов

В работающей системе данные компоненты взаимодействуют следующим образом:

- **Разработчик** определяет объектную модель с помощью Protocol Buffers;
- **Migrator** анализирует модель и создает/обновляет схему БД;
- **Ядро ORM** предоставляет API для работы с объектами;
- **Master** транслирует операции в SQL и взаимодействует с БД;
- **Ядро ORM** преобразует результаты в объекты для использования в приложении.

Эта архитектура обеспечивает полное разделение между объектной моделью приложения и реляционной моделью БД, позволяя каждой развиваться независимо, но сохраняя их синхронизацию.

Методы оптимизации запросов

Эффективность ORM-решения критически зависит от производительности выполнения запросов к базе данных. В разрабатываемой архитектуре будут реализованы следующие методы оптимизации:

1. Объединение запросов (Query Batching)

Объединение запросов позволяет значительно снизить накладные расходы на взаимодействие с БД:

- **N+1 проблема:** автоматическое выявление и решение проблемы N+1 запросов путем замены серии отдельных запросов на один эффективный запрос;
- **Пакетная загрузка связанных объектов:** оптимизация загрузки отношений 1:N и M:N через объединенные запросы с IN-предикатами;
- **Пакетные операции вставки/обновления:** группировка операций изменения данных в одной транзакции с использованием bulk-операций.

Пример оптимизации N+1 проблемы:

```
// Стандартный: N+1 запрос
std::vector<User> users = session.query<User>().all();
for (auto& user : users) {
    // Отдельныйзапросдлякаждогопользователя
    user.roles = session.query<Role>().where("user_id = ?", user.id).all();
}

// Оптимизированный: 2 запроса
std::vector<User> users = session.query<User>().all();
std::vector<int> userIds;
for (const auto& user : users) {
    userIds.push_back(user.id);
}
auto rolesMap = session.query<Role>()
    .where("user_id IN ({})", userIds)
    .groupBy([](const Role& role) { return role.userId; })
    .all();
for (auto& user : users) {
    user.roles = rolesMap[user.id];
}
```

2. Ленивая и жадная загрузка

ORM будет поддерживать различные стратегии загрузки данных:

- **Ленивая загрузка (Lazy Loading):** загрузка связанных объектов только при первом обращении к ним, что уменьшает начальное потребление памяти;

- **Жадная загрузка (Eager Loading):** предварительная загрузка связанных объектов в одном запросе, уменьшающая общее количество обращений к БД;
- **Условная загрузка:** настраиваемые правила для выбора между ленивой и жадной стратегиями в зависимости от контекста.

3. Компиляция запросов

Для максимальной производительности критичных запросов будет использоваться компиляция запросов во время сборки:

- **Статический анализ:** проверка корректности запросов на этапе компиляции;
- **Предкомпиляция:** генерация оптимизированного кода для часто используемых запросов;
- **Параметризованные запросы:** повторное использование скомпилированных планов выполнения.

4. Оптимизация на уровне SQL

Master будет применять ряд оптимизаций при генерации SQL:

- **Минимизация выборки:** выбор только необходимых полей вместо использования SELECT *;
- **Оптимизация JOIN:** анализ и оптимизация порядка и типов соединений таблиц;

Комбинирование всех этих методов оптимизации позволит достичь высокой производительности при сохранении удобства использования ORM. Особое внимание будет уделено оптимизациям, специфичным для C++, таким как эффективное управление памятью, минимизация копирований данных и использование возможностей компиляции.

Заключение

В результате проведенного исследования разработана трехуровневая архитектура ORM для C++, реализующая инверсию зависимостей между объектно-ориентированной и реляционной моделями данных: ядро ORM обеспечивает типобезопасное отображение объектов на таблицы, Master абстрагирует взаимодействие с различными СУБД и оптимизирует генерацию SQL-запросов, а Migrator автоматизирует эволюцию схемы базы данных в соответствии с изменениями в объектной модели. Выбор Protocol Buffers в качестве основы для кодогенерации решает проблему отсутствия встроенной рефлексии в C++, а внедрение разных методов оптимизации обеспечивает высокую производительность, сохраняя при этом простоту использования API — что делает предложенную архитектуру оптимальным решением для высоконагруженных систем, где критичны как строгая типизация и производительность C++, так и удобство работы с реляционными данными.

Список литературы

[1] rezahazegh Understanding Object-Relational Impedance Mismatch — DEV, 2024.

[2] Сагалаев И. Хранение объектов не в БД — 2005.

[3] GeeksForGeeks What is Object-Relational Mapping (ORM) in DBMS? — 2024.

[4] Чепайкин А. Плюсы и минусы написания запросов с ORM и на SQL — Habr, 2025.

[5] Мини-обзор библиотек для Reflection в C++ — Habr, 2015.

[6] Cap’N’Proto Serialization in C++ — wirepair.org, 2023.

[7] @badcasedaily1 Два типа рефлексий в C++ — Habr, 2024.

[8] Elton Minetto JSON vs FlatBuffers vs Protocol Buffers — DEV, 2024.

[9] Никулин А. Apache Thrift RPC Server. Дружим C++ и Java — Habr, 2013.

[10] Александр Фокин Обзор C++26 — cppconf.ru, 2024.

[11] Яндекс YTsaurus — 2024.

Приложения

Дневник студента

Дата	Рабочее место	Краткое содержание выполняемых работ	Отметки руково-дителя
27.02 - 03.03	Удаленно	Изучение проблемы объектно-реляционного несоответствия импеданса. Сбор литературы по теме.	
03.03 - 08.03	Удаленно	Изучение существующих ORM-решений для C++. Анализ их достоинств и недостатков.	
09.03 - 12.03	Удаленно	Исследование возможностей рефлексии в C++. Сравнительный анализ подходов к решению проблемы отсутствия рефлексии.	
12.03 - 18.03	Удаленно	Сравнение технологий кодогенерации: Protocol Buffers, FlatBuffers, Cap’n Proto. Выбор Protocol Buffers для дальнейшей работы.	
19.03 - 28.03	Удаленно	Разработка концепции архитектуры ORM. Проектирование компонентов: Ядро ORM, Master, Migrator.	
28.03 - 10.04	Удаленно	Начало реализации ядра ORM. Разработка базовых классов для отображения объектов на таблицы.	
10.04 - 22.04	Удаленно	Разработка компонента Master для генерации SQL-запросов и работы с различными СУБД.	