Оглавление

Введение **3**

1 Аналитическая часть **4**

1.1 Формализация задачи 4

1.2 Формализация данных 4

1.3 Типы пользователей 6

1.4 Анализ существующих решений 9

1.4.1 Критерии 9

1.4.2 Мобильное приложение горнолыжного курорта «Газпром» 10

1.4.3 Мобильное приложение горнолыжного курорта «Courchevel» 11

1.4.4 Мобильное приложение горнолыжного курорта «Роза Хутор» 12

1.4.5 Сравнение существующих решений 14

1.5 Анализ баз данных 14

1.5.1 Классификация баз данных по месту хранения информации 15

1.5.2 Выбор БД по месту хранения информации 15

2 Конструкторская часть **17**

2.1 Функциональная модель 17

2.2 Проектирование базы данных 17

2.3 Проектирование архитектуры приложения 21

2.4 Алгоритм расчета времени в очереди на подъемник 22

3 Технологическая часть **24**

3.1 Выбор in-memory СУБД 24

3.1.1 Memcached 24

3.1.2 Redis 25

3.1.3 Tarantool 26

3.1.4 Выбор СУБД для решения задачи 27

3.2 Выбор инструментов разработки 28

3.3 Детали реализации 28

3.3.1 Ролевая модель 28
3.3.2 Реализация функции для расчета времени в очереди на подъемник 29
3.3.3 Модели хранения данных 30
3.4 Интерфейс приложения 33

4 Исследовательская часть **40**
4.1 Цель эксперимента 40
4.2 Технические характеристики 40
4.3 Описание эксперимента 40
4.3.1 Сравниваемые алгоритмы 41
4.3.2 Используемые данные 45
4.4 Результаты эксперимента 45

Заключение **48**

Список использованных источников **49**

Введение

Последние исследования показывают, что зимой 2022 года россияне летали на горнолыжные курорты на треть чаще. Эксперты оценивали число купленных билетов на горнолыжные курорты и выяснилось, что российские курорты стали популярнее на 32.6%, а заграничные — на 40% [1].

В связи с этим растет спрос на функциональные и удобные приложения для горнолыжных курортов, которые помогали бы туристам ориентироваться на трассах, планировать свои маршруты по склонам с учетом текущих погодных условий и загруженности подъемников.

Цель курсовой работы – разработка базы данных для онлайн-мониторинга состояния трасс и подъемников горнолыжного курорта.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- проанализировать существующие решения;
- формализовать задачу и данные;
- проанализировать способы хранения данных и системы управления базами данных, выбрать наиболее подходящие решения для поставленной задачи;
- спроектировать и разработать базу данных;
- реализовать программное обеспечение, которое позволит получить доступ к данным посредством REST API [2];
- провести исследование зависимости времени обработки данных от их объема и от распределения вычислений между, базой данных и приложением.

1 Аналитическая часть

В данном разделе приведена формализация задачи и данных, проанализированы существующие решения, рассмотрены типы пользователей и требуемый функционал. Представлен анализ способов хранения данных, а также произведен выбор оптимального для решения поставленной задачи подхода к хранению данных.

1.1 Формализация задачи

Необходимо спроектировать и реализовать базу данных для онлайн-мониторинга состояния трасс и подъемников горнолыжного курорта. Также необходимо разработать интерфейс, позволяющий работать с данной базой для получения и изменения хранящейся в ней информации и мониторинга очередей к подъемникам в онлайн-режиме. Требуется реализовать, как минимум, три вида ролей – пользователь, сотрудник лыжного патруля и администратор.

1.2 Формализация данных

База данных должна хранить информацию о:

- трассах;
- подъемниках;
- связях трасс и подъемников (на одном подъемнике можно добраться до нескольких трасс, и до одной трассы можно добраться на нескольких подъемниках);
- турникетах;
- проездных картах;
- считываниях карт на турникетах подъемников;

- сообщениях о происшествиях;
- пользователях;
- группах пользователей.

В таблице 1.1 приведены категории и сведения о данных.

Таблица 1.1: Категории и сведения о данных

Категория	**Сведения**
Трассы	ID трассы, название трассы, уровень сложности, открытость/закрытость.
Подъемники	ID подъемника, название подъемника, открытость/закрытость, количество мест, время подъема, время в очереди.
Связи трасс и подъемников	ID записи, ID подъемника, ID трассы.
Турникеты	ID турникета, ID подъемника, открытость/закрытость.
Проездные карты	ID карты, дата и время активации, тип.
Считывания карт на турникетах подъемников	ID записи, ID турникета, ID карты, дата и время считывания.
Сообщения о происшествиях	ID сообщения, ID отправителя, ID прочитавшего, текст сообщения.
Пользователи	ID пользователя, ID карты, email (логин), пароль, ID группы пользователей.
Группы пользователей	ID группы пользователей, права доступа.

На рисунке 1.1 отображена ER-диаграмма системы, основанная на приведенной выше таблице.

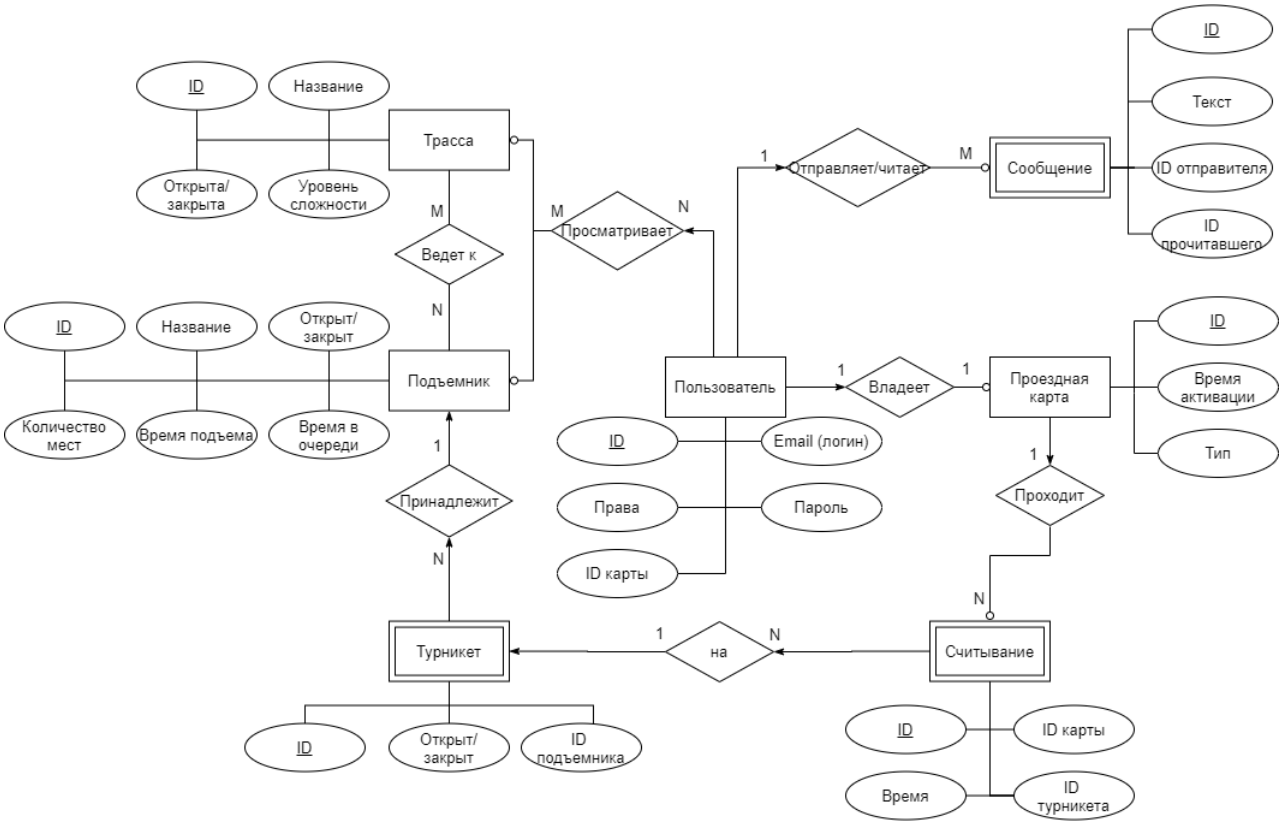


Рис. 1.1: ER-диаграмма

1.3 Типы пользователей

В соответствии с поставленной задачей необходимо разработать приложение с возможностью аутентификации пользователей, что делит их, прежде всего, на авторизованных и неавторизованных. Для управления приложением необходима ролевая модель: авторизованный (обычный) пользователь, сотрудник лыжного патруля и администратор.

Для каждого типа пользователя предусмотрен свой набор функций, который можно описать с помощью текста и Use Case Diagram (диаграммы прецедентов). Она состоит из графической диаграммы, описывающей действующие лица и прецеденты – конкретные действия, которые выполняет пользователь при работе с системой.

Набор функций неавторизованного пользователя:

- регистрация,
- аутентификация,
- просмотр информации о состоянии трасс и подъемников,
- просмотр информации о связях трасс и подъемников.

На рисунке 1.2 представлена диаграмма прецедентов для неавторизованного пользователя.

Рис. 1.2: Диаграмма прецедентов для неавторизованного пользователя

Набор функций авторизованного пользователя:

- выход,
- просмотр информации о состоянии трасс и подъемников,
- просмотр информации о связях трасс и подъемников,
- отправка сообщений о происшествиях.

На рисунке 1.3 представлена диаграмма прецедентов для неавторизованного пользователя.

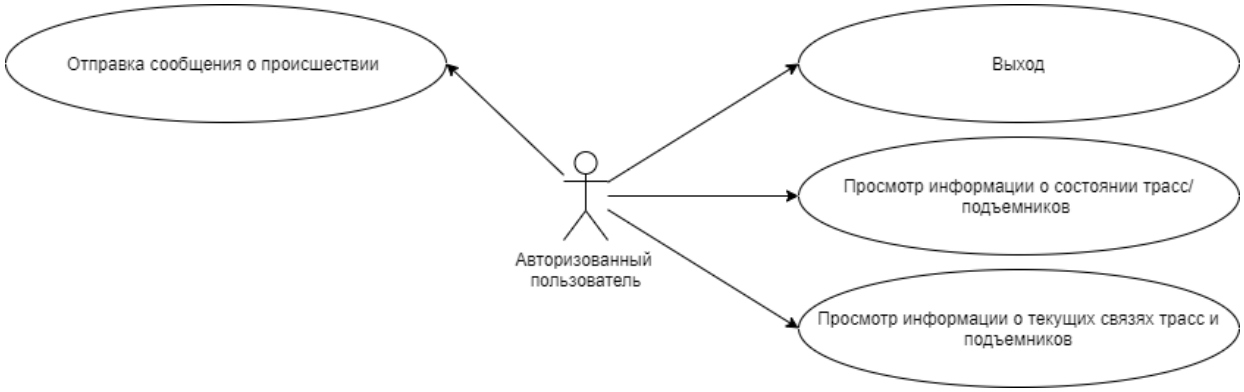


Рис. 1.3: Диаграмма прецедентов для авторизованного пользователя

Набор функций сотрудника лыжного патруля:

- выход,
- просмотр и изменение информации о состоянии трасс и подъемников,
- просмотр и изменение информации о связях трасс и подъемников,
- просмотр сообщений о происшествиях.

На рисунке 1.4 представлена диаграмма прецедентов для сотрудника лыжного патруля.

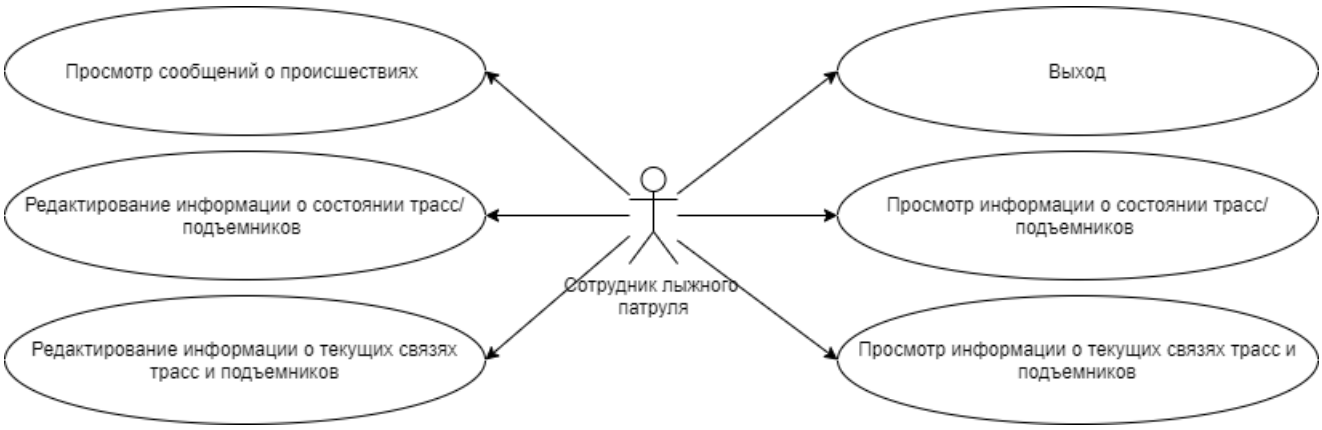


Рис. 1.4: Диаграмма прецедентов для сотрудника лыжного патруля

Набор функций администратора:

- выход,
- просмотр и изменение всей информации, доступной в базе данных, в том числе права доступа групп и отдельных пользователей.

На рисунке 1.5 представлена диаграмма прецедентов для администратора.

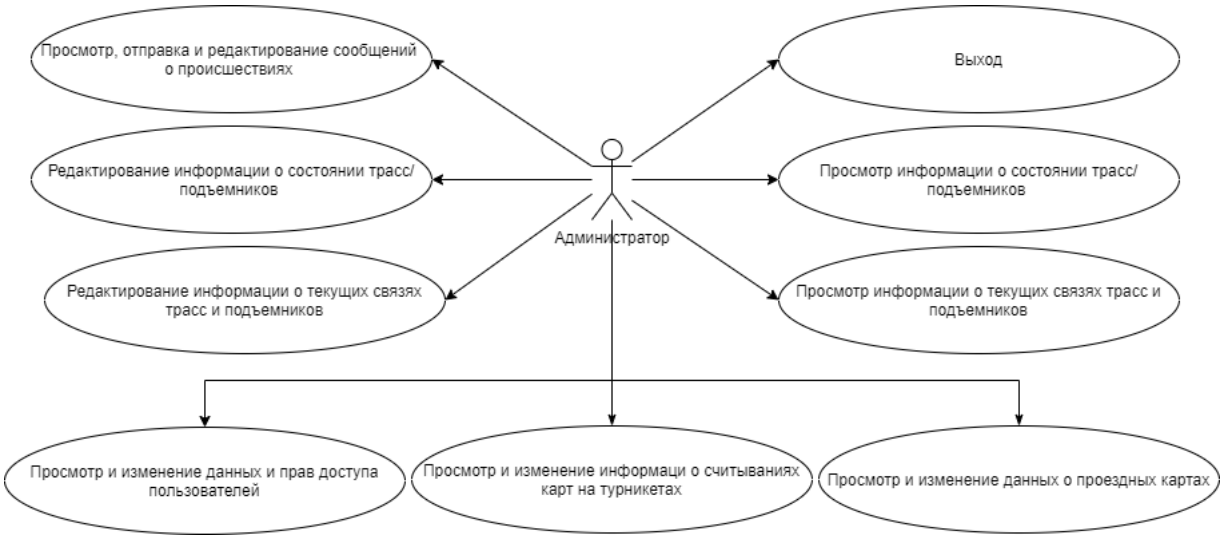


Рис. 1.5: Диаграмма прецедентов для администратора

1.4 Анализ существующих решений

1.4.1 Критерии

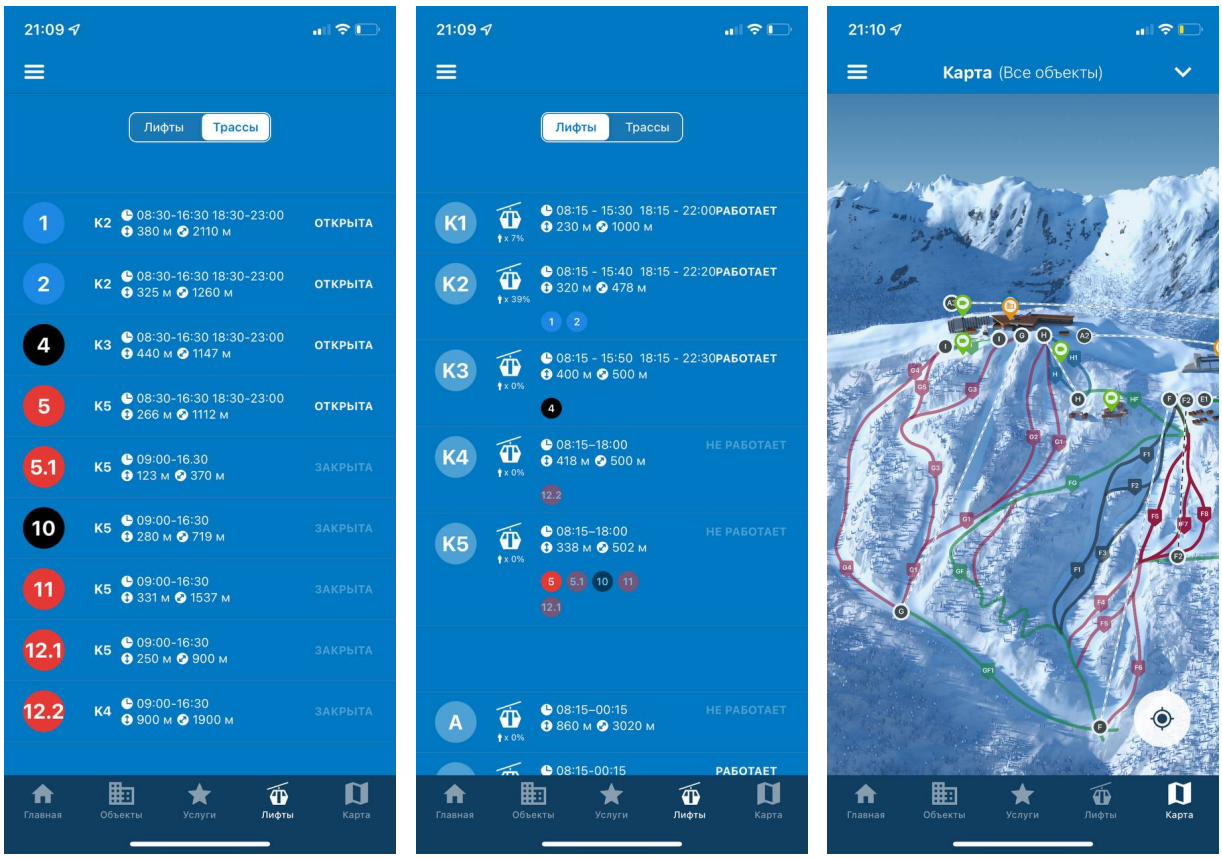
Проведем анализ аналогичных решений по следующим критериям:

1. наличие информации об открытости/закрытости трасс и подъемников;
2. наличие информации об очередях на подъемниках;
3. наличие сводной информации о связях трасс и подъемников;

4. возможность получить информацию о том, до каких трасс можно добраться на конкретном подъемнике/на каких подъемниках можно добраться до конкретной трассы.

1.4.2 Мобильное приложение горнолыжного курорта «Газпром»

В приложении горнолыжного курорта «Газпром» есть возможность просмотреть информацию об открытости/закрытости трасс (рисунок 1.6, a) и подъемников (рисунок 1.6, b), а также сводную информацию о связях трасс и подъемников с помощью карты курорта (рисунок 1.6, c):



(a) (b) (c)

Рис. 1.6: Просмотр информации в мобильном приложении «Газпром»

1.4.3 Мобильное приложение горнолыжного курорта «Courchevel»

В приложении горнолыжного курорта «Courchevel» на единой карте можно просмотреть информацию об открытости/закрытости трасс и подъемников, а также сводную информацию о связях трасс и подъемников (рисунок 1.7):

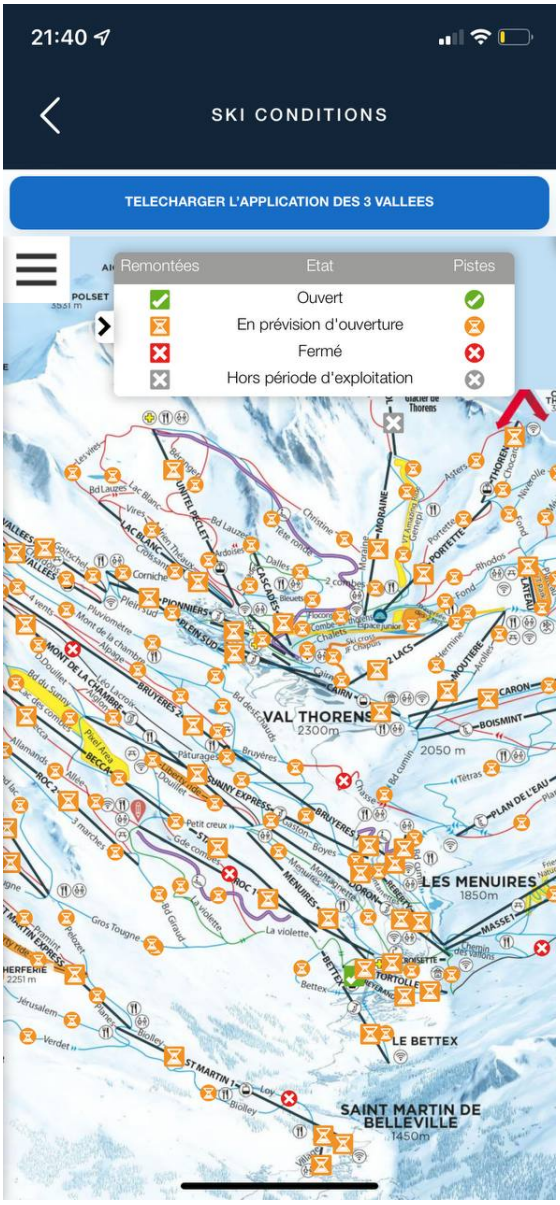
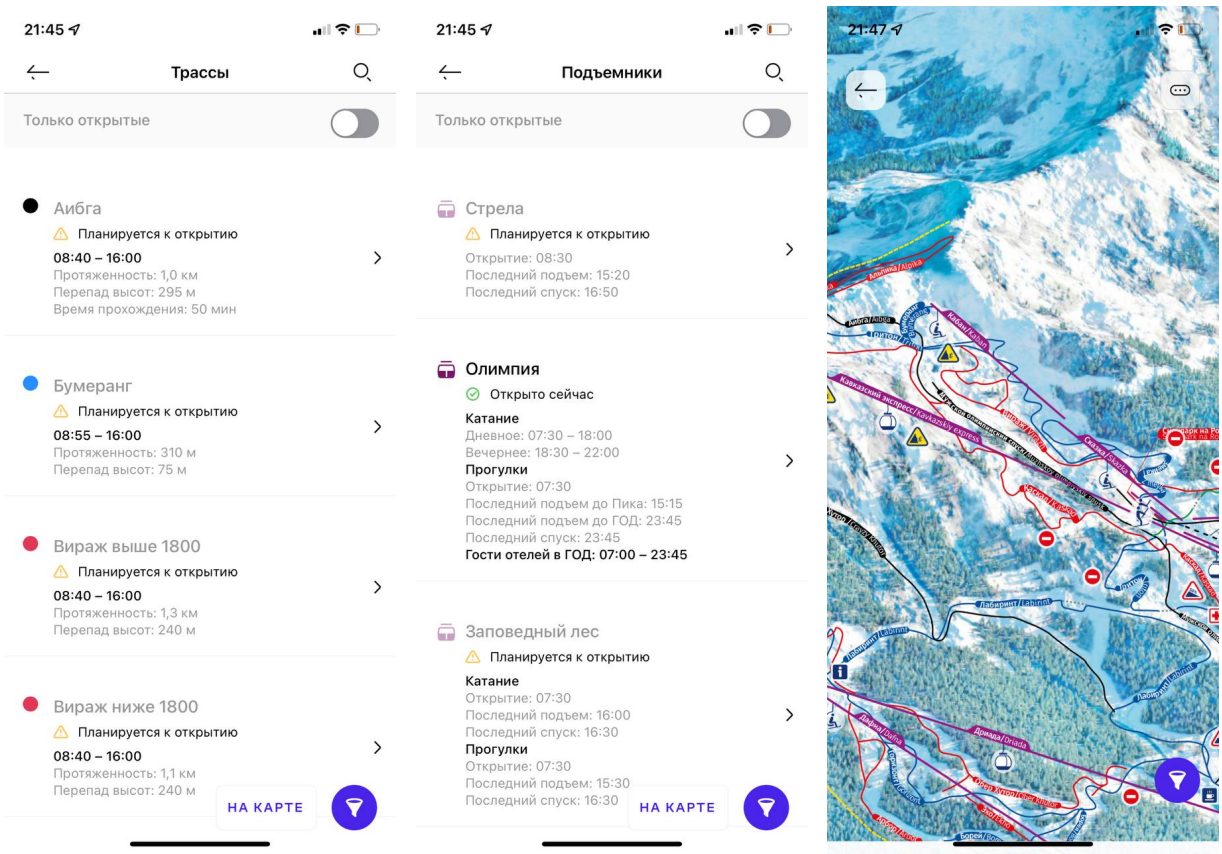


Рис. 1.7: Просмотр информации в мобильном приложении «Courchevel»

1.4.4 Мобильное приложение горнолыжного курорта «Роза Хутор»

В приложении горнолыжного курорта «Газпром» есть возможность просмотреть информацию об открытости/закрытости трасс (рисунок 1.8, a) и подъемников (рисунок 1.8, b), а также сводную информацию о связях трасс и подъемников с помощью карты курорта (рисунок 1.8, c):



(a) (b) (c)

Рис. 1.8: Просмотр информации в мобильном приложении «Роза Хутор»

Также есть возможность в онлайн-режиме просматривать камеры, расположенные на различных объектах курорта (пример показан на рисунке 1.9). Этим можно воспользоваться для мониторинга очередей на подъемниках.

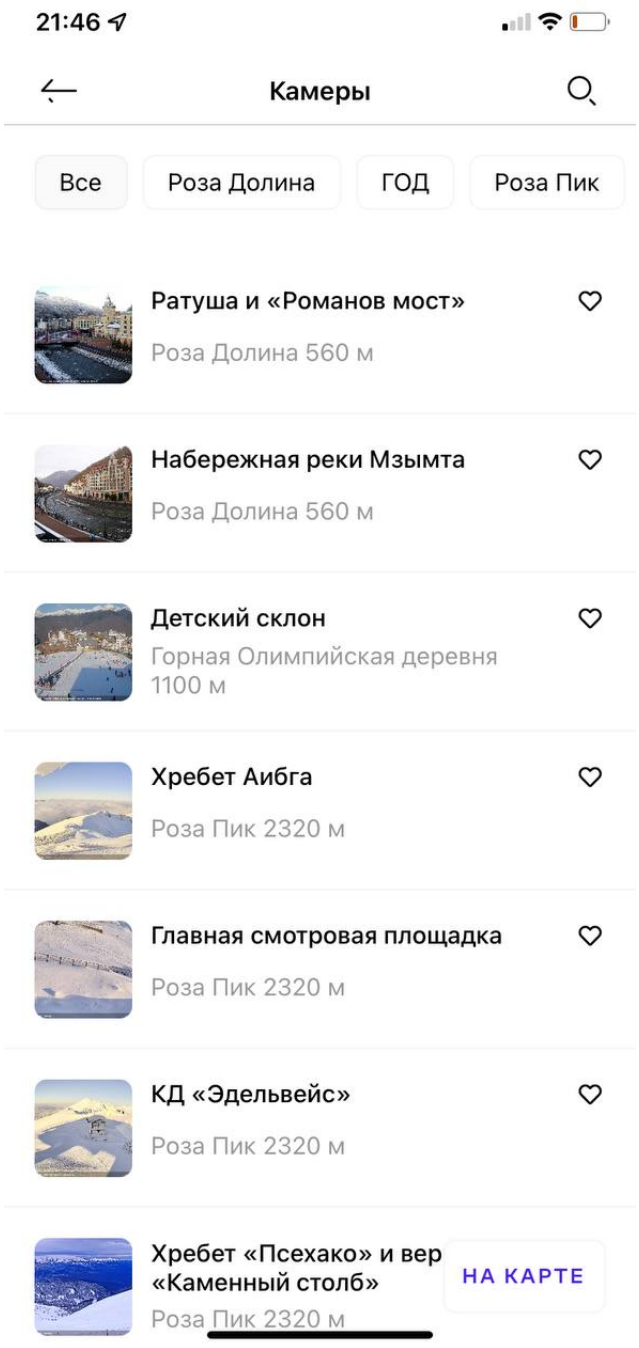


Рис. 1.9: Онлайн-просмотр камер в приложении «Роза Хутор»

1.4.5 Сравнение существующих решений

В таблице 1.2 приведено сравнение решений по наличию в них информации, перечисленной в пункте 1.4.1. В таблице приняты следующие обозначения: открытость – информация об открытости/закрытости трасс и подъемников, очереди – информации об очередях на подъемниках, связи (св.) – сводная информация о связях трасс и подъемников, связи (конкр.) – информация о связях конкретной трассы с подъемниками или наоборот.

Таблица 1.2: Сравнение решений

Название	Открытость	Очереди	Связи (св.)	Связи (конкр.)
Газпром	+ (список)	-	+ (на карте)	-
Courchevel	+ (на карте)	-	+ (на карте)	-
Роза Хутор	+ (список)	+ (камеры)	+ (на карте)	-

Таким образом, ни одно из приложений не отображает всю информацию, которую предполагается предоставлять в разрабатываемом приложении. В частности, наименее представлена информация об очередях на подъемниках, которая позволила бы посетителям выбирать мене загруженные объекты и тем самым делать нагрузку более равномерной. Предоставление же информации о том, как добраться до конкретной трассы позволит пользователям (в особенности, новым) упростить организацию своего катания.

1.5 Анализ баз данных

Для решения поставленной цели необходимо разработать базу данных (БД). БД – это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе[3].

1.5.1 Классификация баз данных по месту хранения информации

По месту хранения информации БД можно разделить на [4]:

- традиционные, которые хранят информацию на жестком диске или другом постоянном носителе;
- in-memory databases (IMDB) (резидентные базы данных), которые хранят информацию непосредственно в оперативной памяти.

IMDB появились как ответ традиционным БД в связи со снижением стоимости оперативной памяти. Хранение данных в этой области позволяет увеличить скорость их обработки в сотни раз [5]. При этом резидентные БД обеспечивают высокую пропускную способность систем, критичных к производительности [6].

Обратной стороной этих достоинств являются следующие недостатки:

- однопоточность и эффективная утилизация только одного ядра процессора, что не позволяет в полной мере воспользоваться возможностями современных многоядерных серверов;
- энергозависимость и привязка к размеру оперативной памяти.

В практическом плане IMDB-системы особенно востребованы в тех приложениях работы с данными в реальном времени, где требуется минимальное время отклика [7].

1.5.2 Выбор БД по месту хранения информации

Основным требованием к разрабатываемой БД является предоставление возможности **онлайн**-мониторинга состояния объектов горнолыжного курорта. Задача предполагает постоянное добавление и изменение данных. С особо высокой частотой будут добавляться считывания карт на турникетах подъемников. При этом время в очереди на подъемник должно регулярно пересчитываться и обновляться, чтобы пользователь не получил

устаревшую информацию. Решение также предполагает быструю отзывчивость на запросы пользователя.

Таким образом, задача является типовым примером использования IMDB. И поскольку в современных in-memory СУБД существуют надежные и достаточно простые способы устранения указанных недостатков этих БД, было принято решение использовать именно этот подход к хранению данных.

Вывод

В данном разделе была проведена формализация задачи и данных, проанализированы существующие решения, рассмотрены типы пользователей и требуемый функционал. В результате анализа способов хранения данных, для данной задачи был сделан выбор в пользу резидентной БД.

2 Конструкторская часть

В данном разделе представлены этапы проектирования системы: функциональная модель, диаграмма базы данных и архитектуры приложения.

2.1 Функциональная модель

На рисунке 2.1 изображена функциональная модель, отображающая структуру и функции системы.

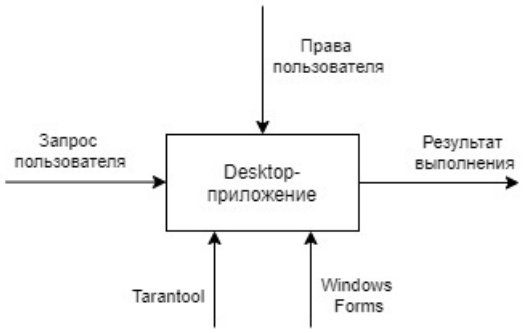


Рис. 2.1: Функциональная модель приложения

2.2 Проектирование базы данных

В соответствии с ER-диаграммой системы, изображенной на рисунке 1.1, база данных должна хранить следующие таблицы:

- таблица трасс slopes;
- таблица подъемников lifts;
- таблица связей трасс и подъемников lifts_slopes;
- таблица турникетов turnstiles;
- таблица проездных карт cards;
- таблица считываний карт на турникетах подъемников card_readings;

- таблица сообщений о происшествиях messages;
- таблица пользователей users.

На рисунке 2.2 представлена диаграмма разрабатываемой базы данных.

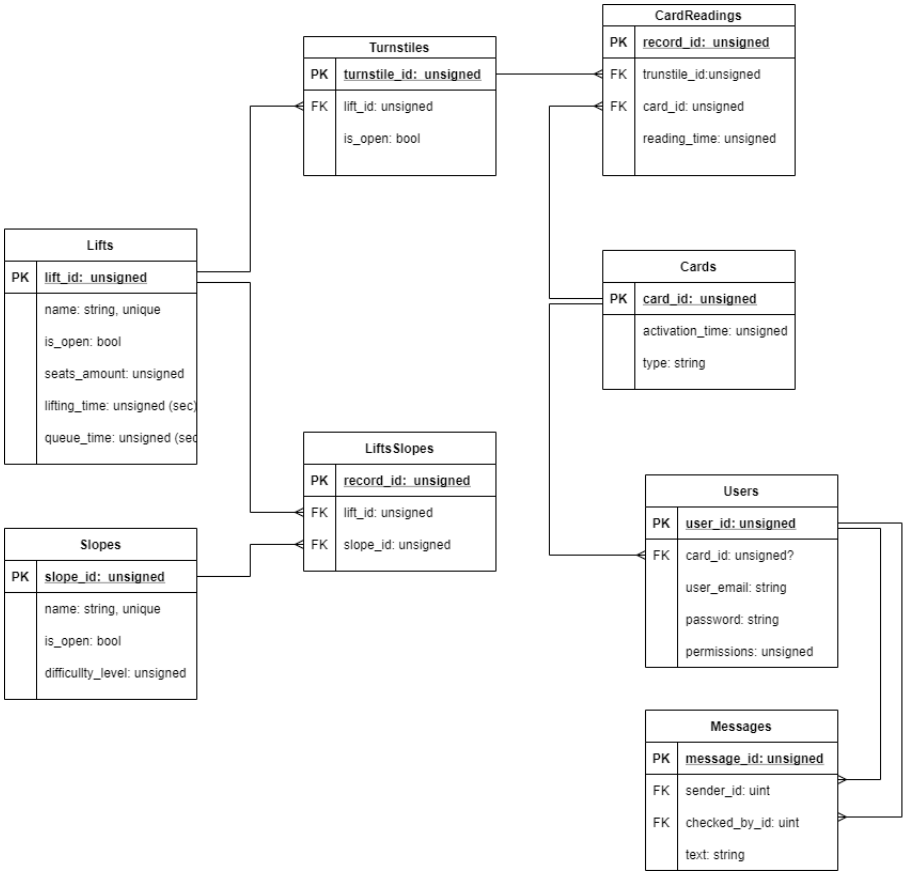


Рис. 2.2: Диаграмма базы данных

Таблица slopes хранит информацию о трассах и содержит следующие поля:

- slope_id – уникальный идентификатор трассы, PK (первичный ключ, англ. primary key), unsigned;
- slope_name – уникальное название, string;
- difficulty_level – уровень сложности, unsigned;
- is_open – открыта или закрыта, boolean.

Таблица lifts хранит информацию о подъемниках и содержит следующие поля:

- lift_id – уникальный идентификатор подъемника, PK, unsigned;
- lift_name – уникальное название, string;
- is_open – открыт или закрыт, boolean;
- seats_amount – количество мест, unsigned;
- lifting_time – время подъема в секундах, unsigned;
- queue_time – время в очереди в секундах, unsigned;

Таблицы slopes и lifts связаны отношением многие-ко-многим. Таблица lifts_slopes хранит информацию об этом отношении (связи трасс и подъемников) и содержит следующие поля:

- record_id – уникальный идентификатор записи, PK, unsigned;
- lift_id – идентификатор подъемника, FK на поле lift_id таблицы lifts, unsigned;
- slope_id – идентификатор трассы, FK на поле slope_id таблицы slopes, unsigned.

Таблица turnstiles хранит информацию о турникетах и содержит следующие поля:

- turnstile_id – уникальный идентификатор турникета, PK, unsigned;
- lift_id – идентификатор подъемника, на котором установлен этот турникет, FK на поле lift_id таблицы lifts, unsigned;
- is_open – открыт или закрыт, boolean.

Таблица cards хранит информацию о проездных картах и содержит следующие поля:

- card_id – уникальный идентификатор карты, PK, unsigned;
- activation_time – дата и время активации в формате Unix-time (количество секунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года), unsigned;

- type – тип карты, string.

Таблица card_readings хранит информацию о считываниях карт на турникетах подъемников и содержит следующие поля:

- record_id – уникальный идентификатор считывания, PK, unsigned;
- turnstile_id – идентификатор турникета, на котором произошло считывание, FK на поле turnstile_id таблицы turnstiles, unsigned;
- card_id – идентификатор проездной карты, которая прошла считывание, FK на поле card_id таблицы cards, unsigned;
- reading_time – дата и время считывания в формате Unix-time, unsigned.

Таблица users хранит информацию о пользователях и содержит следующие поля:

- user_id – уникальный идентификатор пользователя, PK, unsigned;
- card_id – идентификатор проездной карты, которая принадлежит пользователю (может отсутствовать), FK на поле card_id таблицы cards, unsigned;
- user_email – адрес электронной почты (он же будет использоваться как логин), string;
- password – пароль, string;
- permissions – права доступа (роль), unsigned.

Таблица messages хранит информацию о сообщениях пользователей о происшествиях и содержит следующие поля:

- message_id – уникальный идентификатор сообщения, PK, unsigned;
- sender_id – идентификатор отправителя, FK на поле user_id таблицы usres, unsigned;
- checked_by_id – идентификатор прочитавшего, FK на поле user_id таблицы usres, unsigned;
- text – текст сообщения, string.

2.3 Проектирование архитектуры приложения

Интерфейс для работы с базой данных представляет собой многокомпонентное desktop-приложение. На базовом уровне выделены три компонента: компонент доступа к данным, компонент бизнес-логики и компонент реализации пользовательского интерфейса. На рисунке 2.3 представлена архитектура приложения.

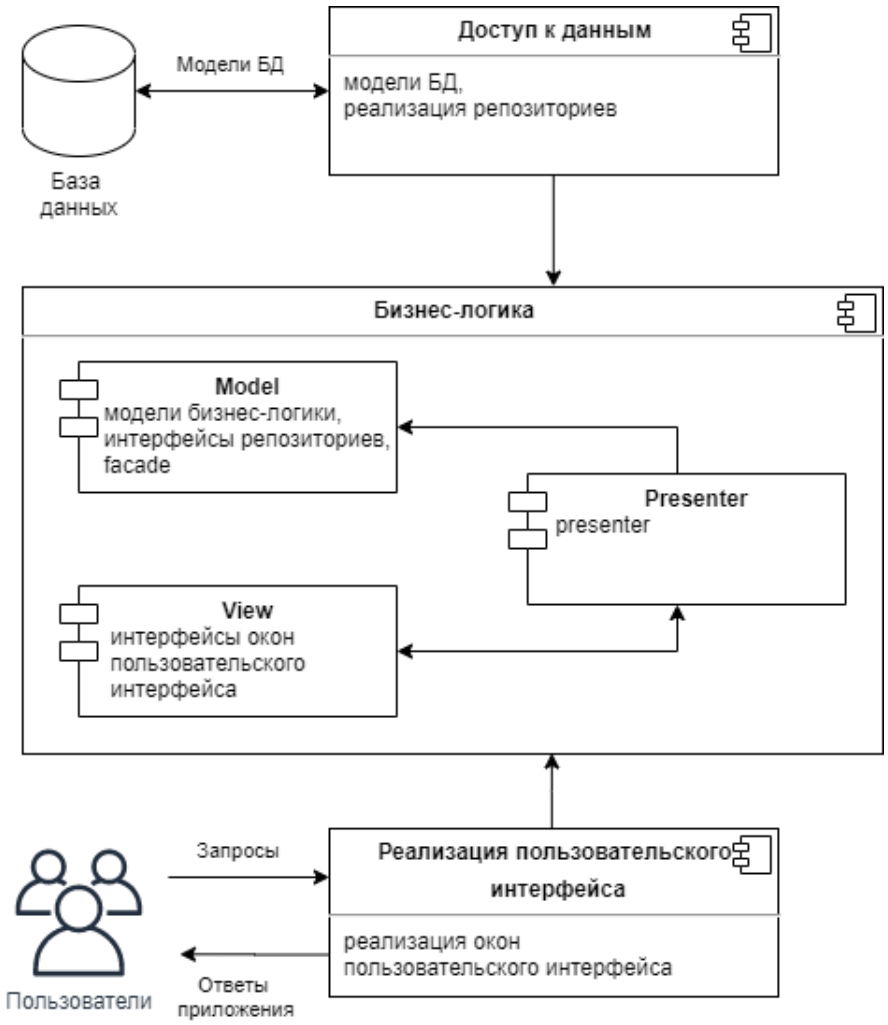


Рис. 2.3: Архитектура приложения

Компонент бизнес-логики спроектирован по схеме MVP (Model-View-Presenter) и является главным (независимым). Он отвечает за логику обработки запросов и данных.

Компонент доступа к данным отвечает за получение данных из БД, их

изменение, добавление и удаление. Для его реализации использован паттерн «репозиторий».

Компонент реализации пользовательского интерфейса предоставляет реализацию UI для выбранного технологического стека.

2.4 Алгоритм расчета времени в очереди на подъемник

Для онлайн-мониторинга очередей на подъемниках необходимо реализовать функцию update_queue_time, которая будет обновлять время в очереди к каждому из них. Эта функция будет периодически, через некоторый заданный интервал времени, вызываться из приложения для каждого подъемника из БД.

На рисунке 2.4 представлены схемы алгоритмов функции update_queue_time и вспомогательной функции count_card_readings, которая возвращает количество считываний на указанном подъемнике в заданный промежуток времени.

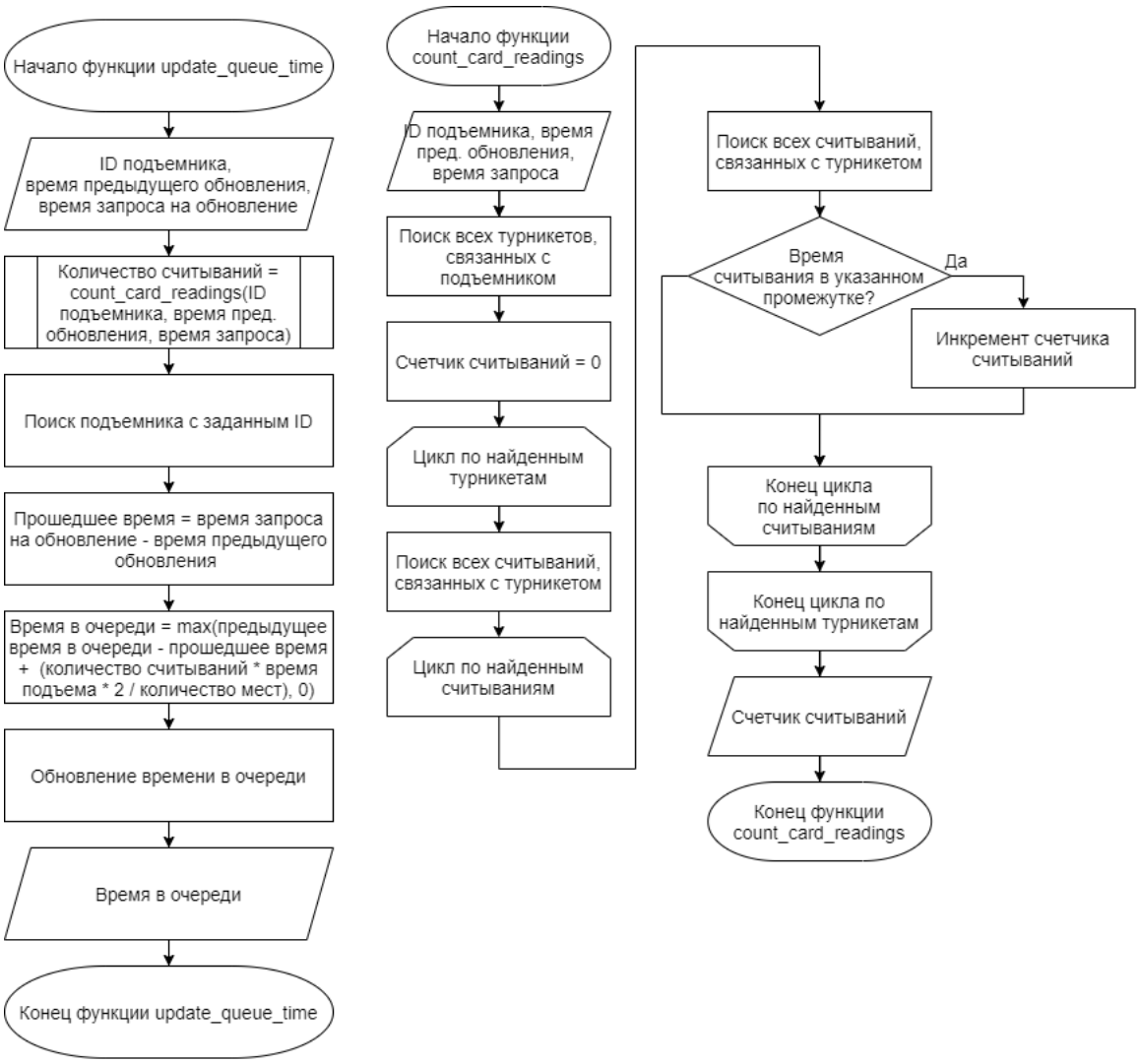


Рис. 2.4: Схема работы алгоритма функции update_queue_time

Вывод

В данном разделе была приведена функциональная модель системы, спроектирована база данных и архитектура приложения, приведена схема работы алгоритма расчета времени в очереди на подъемниках.

3 Технологическая часть

В данном разделе проводится анализ in-memory СУБД и выбор наиболее подходящей для решения поставленной задачи, производится выбор инструментов разработки, приводятся детали реализации и интерфейс приложения.

3.1 Выбор in-memory СУБД

Система управления базами данных (СУБД) – это совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных[3].

В пункте 1.5.2 среди подходов к хранению данных был сделан выбор в пользу IMDB-систем, поэтому далее будет приведен обзор in-memory СУБД и произведен выбор наиболее подходящей для поставленной задачи.

3.1.1 Memcached

Memcached - это высокопроизводительная система кэширования данных в оперативной памяти, предназначенная для использования в ускорении динамических веб-приложений за счет уменьшения нагрузки на базу данных. Memcached относится к семейству решений для управления данными NoSQL и основана на модели данных с ключевыми значениями [8].

Данная СУБД спроектирована так, чтобы все операции имели алгоритмическую сложность O(1), то есть время выполнения любой операции не зависит от количества хранящихся ключей. Это означает, что некоторые операции или возможности, реализация которых требует всего лишь линейного (O(n)) времени, в ней отсутствуют. Так, например, в Memcached отсутствует возможность группировки ключей.

Memcached не является надежным хранилищем – возможна ситуация, когда ключ будет удален из кэша раньше окончания его срока жизни.

Управление внутренней памятью Memcached более эффективно в простейших случаях использования (при кэшировании относительно неболь-

ших и статических данных), поскольку оно потребляет сравнительно мало ресурсов памяти для метаданных. Строки (единственный тип данных, поддерживаемый Memcached) идеально подходят для хранения данных, которые только читаются, потому что строки не требуют дальнейшей обработки. Тем не менее, эффективность управления памятью Memcached быстро уменьшается, когда размер данных является динамическим, после чего память Memcached может стать фрагментированной [9].

Еще одно преимущество Memcached – достаточно простая масштабируемость: поскольку данная система многопоточна, ее можно увеличить, просто предоставив больше вычислительных ресурсов. Однако это может привести к потере части или всех кэшированных данных (в зависимости от того, используется ли постоянное хеширование).

3.1.2 Redis

Redis [10] – резидентная система управлениями базами данных класса NoSQL с открытым исходным кодом.

Основной структурой данных, с которой работает Redis является структура типа «ключ-значение», причем значения могут быть пяти различных типов. Данная СУБД используется как для хранения данных, так и для реализации кэшей и брокеров сообщений.

Redis хранит данные в оперативной памяти и снабжена механизмом «снимков» и журналирования, что обеспечивает постоянное хранение данных. Существует поддержка репликации данных типа master-slave, транзакций и пакетной обработки комманд.

Redis позволяет осуществлять мелкомасштабный контроль за вытеснением данных, предоставляя выбор из шести различных политик вытеснения [11].

К недостаткам Redis можно отнести отсутствие вторичных индексов и триггеров. Также транзакции в данной СУБД не удовлетворяют свойствам ACID (Atomicity – Атомарность, Consistency – Согласованность, Isolation – Изолированность, Durability – Долговечность) [12].

3.1.3 Tarantool

Tarantool [13] – это платформа in-memory вычислений с гибкой схемой хранения данных для эффективного создания высоконагруженных приложений. Включает себя базу данных и сервер приложений на языке программирования Lua [14].

Записи в Tarantool хранятся в пространствах (space) – аналог таблицы в реляционной базе данных SQL. Внутри пространства находятся кортежи (tuples), которые похожи на строку в таблице SQL.

Tarantool объединяет в себе преимущества, характерные для кэша:

- «горячие данные»;
- оптимальная работа при высокой параллельной нагрузке;
- низкая задержка (99% запросов $<$ 1 мс, 99,9% запросов $<$ 3 мс);
- поддерживаемая загрузка на запись — до 1 миллиона транзакций в секунду на одном ядре ЦПУ;
- система работает постоянно, не нужно делать перерыв на профилактические работы,

и достоинства классических СУБД:

- персистентность;
- транзакции со свойствами ACID;
- наличие репликации (master-slave и master-master);
- наличие хранимых процедуры.
- поддержка первичных и вторичных индексов (в том числе, составных).

В Tarantool реализован механизм «снимков» текущего состояния хранилища и журналирования всех операций, что позволяет восстановить состояние базы данных после ее перезагрузки.

К недостаткам данной СУБД можно отнести относительно малое количество поддерживаемых языков (C, C#, C++, Erlang, Go, Java, JavaScript, Lua, Perl, PHP, Python, Rust) [13], а также более высокий порог входа по сравнению с ранее рассмотренными СУБД [11].

3.1.4 Выбор СУБД для решения задачи

В данной работе не предполагается хранение в БД очень большого количества информации, и в этом случае требованию о коротком времени отклика удовлетворяют все рассмотренные СУБД.

Рассмотрим несколько требований, которым должна удовлетворять СУБД для решения поставленной задачи.

1. Для удобного хранения данных, указанных в пункте 1.2, СУБД должна предоставлять, как минимум , два типа данных – строки и целые числа.

2. Для предоставления пользователям перечисленных в пункте 1.3 функций, в СУБД должна быть реализована возможность создания вторичных индексов, а также триггеров или хранимых процедур для выполнения сложных вычислений.

3. Необходимо надежное хранение данных, без риска их потери даже в случае сбоя в системе (то есть поддержка репликации данных).

В таблице 3.1 приведено сравнение рассмотренных СУБД по перечисленным критериям.

Таблица 3.1: Сравнение in-memory СУБД

Название	Требование 1	Требование 2	Требование 3
Memcached	-	-	-
Redis	+	-	+
Tarantool	+	+	+

Из рассмотренных in-memory СУБД всем перечисленным требованиям удовлетворяет Tarantool, поэтому именно он был выбран для использования в данной работе.

3.2 Выбор инструментов разработки

В качестве языка программирования выбран язык C# [15]. Во-первых, он объектно-ориентирован, что позволит использовать наследование, интерфейсы, абстракции и так далее. Во-вторых, C# имеет коннектор progaudi.tarantool [16] для платформы in-memory вычислений Tarantool.

В качестве среды разработки была выбрана «Microsoft Visual Studio 2022» [17], поскольку она:

1) имеет множество удобств для написания и отладки кода;

2) является бесплатной для студентов;

3) обеспечивает работу с фреймворком Windows Forms[18], который был выбран для реализации пользовательского интерфейса из-за своей простоты в реализации.

Для упаковки приложения в готовый продукт была выбрана система контейнеризации Docker [19]. С его помощью можно создать изолированную среду для программного обеспечения, которое можно будет развернуть на различных устройствах без дополнительных настроек.

Тестирование программного продукта производилось с помощью фреймворка xUnit [20]. Данный фреймворк позволяет писать функциональные тесты.

3.3 Детали реализации

3.3.1 Ролевая модель

Необходимо реализовать ролевую модель в соответствии с пунктом 1.3. Роль – это разрешение, предоставляемое группе пользователей для доступа к данным.

Ролевая модель реализована на нескольких уровнях, в том числе, на уровне базы данных, что продемонстрировано в листинге 3.1. Последова-

тельно создаются и наделяются правами роли неавторизованного пользователя, авторизованного пользователя, сотрудника лыжного патруля и администратора, каждый наследует и расширяет права предыдущего.

```
box.schema.role.create('unauthorized_user')
box.schema.role.grant('unauthorized_user', 'read', 'space', 'lifts', {if_not_exists=true})
box.schema.role.grant('unauthorized_user', 'read', 'space', 'slopes', {if_not_exists=true})
box.schema.role.grant('unauthorized_user', 'read', 'space', 'lifts_slopes', {if_not_exists=true})


box.schema.role.create('authorized_user')
box.schema.role.grant('authorized_user', 'unauthorized_user')
box.schema.role.grant('authorized_user', 'read,write', 'space', 'messages', {if_not_exists=true})


box.schema.role.create('ski_patrol')
box.schema.role.grant('ski_patrol', 'authorized_user')
box.schema.role.grant('unauthorized_user', 'write', 'space', 'lifts', {if_not_exists=true})
box.schema.role.grant('unauthorized_user', 'write', 'space', 'slopes', {if_not_exists=true})
box.schema.role.grant('unauthorized_user', 'write', 'space', 'lifts_slopes', {if_not_exists=true})

box.schema.role.create('ski_admin')
box.schema.role.grant('ski_admin', 'read,write,execute,create,alter,drop', 'universe')
```

Листинг 3.1: Ролевая модель на уровне БД

3.3.2 Реализация функции для расчета времени в очереди на подъемник

В листинге 3.2 представлены реализации функций update_queue_time и count_card_readings, предназначенных для обновлениия время в очереди

к подъемнику. Алгоритм работы этих функций был описан в пункте 2.4.

```
function count_card_readings(lift_id, date_from, date_query)
  connected_turnstiles = turnstiles.index.index_lift_id:select({lift_id})
  counter = 0
  for k,v in pairs(connected_turnstiles) do
    cur_turnstile_id = v["turnstile_id"]
    card_readings_on_turnstile = card_readings.index.index_turnstile:select({
    cur_turnstile_id})

    for k,v in pairs(card_readings_on_turnstile) do
      if (v["reading_time"] >= date_from and v["reading_time"] < date_query)
    then
        counter = counter + 1
      end
    end
  end
  return counter
end

function update_queue_time(lift_id, date_from, date_query)
  card_readings_amount = count_card_readings(lift_id, date_from, date_query)
  lift = lifts:get{lift_id}
  time_delta = date_query - date_from
  new_queue_time = math_module.max(math_module.ceil(
      lift["queue_time"] - time_delta +
      (card_readings_amount * lift["lifting_time"] * 2 / lift["seats_amount"])
    ), 0)
  lifts:update(lift_id, {{'=', 6, new_queue_time}})
  return new_queue_time
end
```

Листинг 3.2: Функции update_queue_time и count_card_readings

3.3.3 Модели хранения данных

На примере таблицы спусков slopes рассмотрим модель хранения данных и реализацию доступа к данным.

В листинге 3.3 продемонстрировано создание спейса (таблицы) slopes в БД. Особенность Tarantool заключается в том, что для поиска кортежа (строки) в спейсе по значению его поля, это поле должно быть проиндексировано. Так как в спейсе slopes будет необходимо искать кортежи как по значению поля slope_id (PK), так и по значению поля slope_name, то для

этого спейса необходимо создать два соответствующих индекса (первичный и вторичный).

```
slopes = box.schema.space.create('slopes', {field_count=4})
slopes:format({
    {name = 'slope_id', type = 'unsigned'},
    {name = 'slope_name', type = 'string'},
    {name = 'is_open', type = 'boolean'},
    {name = 'difficulty_level', type = 'unsigned'}})
slopes:create_index('primary')
slopes:create_index('index_name', {parts = {'slope_name'}})
```

Листинг 3.3: Создание спейса slopes

В листинге 3.4 приведена реализация модели slopes, используемой в компоненте бизнес-логики. Отношение между таблицами slopes и lifts многие-ко-многим здесь реализована с помощью хранения связанных со спуском подъемников в модели спусков, и наоборот. Связанные объекты добавляются в модель по запросу.

```
public record class Slope {
  public uint SlopeID { get; }
  public string SlopeName { get;}
  public bool IsOpen { get; }
  public uint DifficultyLevel { get; }
  public List<Lift>? ConnectedLifts { get; }

  public Slope(uint slopeID, string slopeName, bool isOpen, uint
   difficultyLevel){
    this.SlopeID = slopeID;
    this.SlopeName = slopeName;
    this.IsOpen = isOpen;
    this.DifficultyLevel = difficultyLevel;}
  public Slope(Slope slope, List<Lift> connectedLifts){
    this.SlopeID = slope.SlopeID;
    this.SlopeName = slope.SlopeName;
    this.IsOpen = slope.IsOpen;
    this.DifficultyLevel = slope.DifficultyLevel;
    this.ConnectedLifts = connectedLifts;}
}
```

Листинг 3.4: Модель базы данных на примере таблицы slopes

В листинге 3.5 приведена часть реализации репозитория TarantoolSlopesRepository, который обеспечивает доступ к данным спейса slopes из БД. Отражены функция GetSlopeByNameAsync, осуществляющая поиск спуска по его названию с помощью ранее созданного вторичного индекса, а также функция AddSlopeAutoIncrementAsync, которая осуществляет вставку нового кортежа, используя первичный ключ с автоматическим увеличением.

```
1  public class TarantoolSlopesRepository : ISlopesRepository
2  {
3    private IIndex _indexPrimary;
4    private IIndex _indexName;
5    private ISpace _space;
6    private IBox _box;
7    ...
8
9    public async Task<Slope> GetSlopeByNameAsync(string name)
10   {
11     var data = await _indexName.Select<ValueTuple<string>,
     SlopeDB>
12     (ValueTuple.Create(name));
13
14     if (data.Data.Length != 1)
15     {
16       throw new SlopeNotFoundException($"Error: couldn't find
     slope with name={name}");
17     }
18
19     return SlopeConverter.DBToBL(data.Data[0]);
20   }
21
22   public async Task<uint> AddSlopeAutoIncrementAsync(string
     slopeName, bool isOpen, uint difficultyLevel)
23   {
24     try
25     {
26       var result = await _box.Call_1_6<SlopeDBNoIndex,
     SlopeDB>("auto_increment_slopes", (new
     SlopeDBNoIndex(slopeName, isOpen, difficultyLevel)));
27       return SlopeConverter.DBToBL(result.Data[0]).SlopeID;
28     }
```

```
29      catch (Exception ex)
30      {
31          throw new SlopeAddAutoIncrementException();
32      }
33    }
34    ...
35 }
```

Листинг 3.5: Часть реализации репозитория для доступа к данным спейса slopes из БД

В используемом коннекторе progaudi.tarantool не реализована соответствующая функция «space_object:auto_increment()» СУБД Tarantool, поэтому на уровне базы данных для каждого спейса реализованы обертки для данной функции и конкретного спейса. Именно они вызываются из приложения, что продемонстрировано в функции AddSlopeAutoIncrementAsync в листинге выше. Пример такой функции-обертки на уровне БД приведен в листинге 3.6.

```
1 function auto_increment_slopes(slope_name, is_open,
    difficulty_level)
2     return box.space.slopes:auto_increment{slope_name, is_open,
    difficulty_level}
3 end
```

Листинг 3.6: Обертка для функции «space_object:auto_increment()» и спейса slopes

3.4 Интерфейс приложения

При запуске приложения пользователь считается неавторизованным. Открывается главное окно, продемонстрированное на рисунке 3.1, в котором размещены кнопки для открытия других окон.

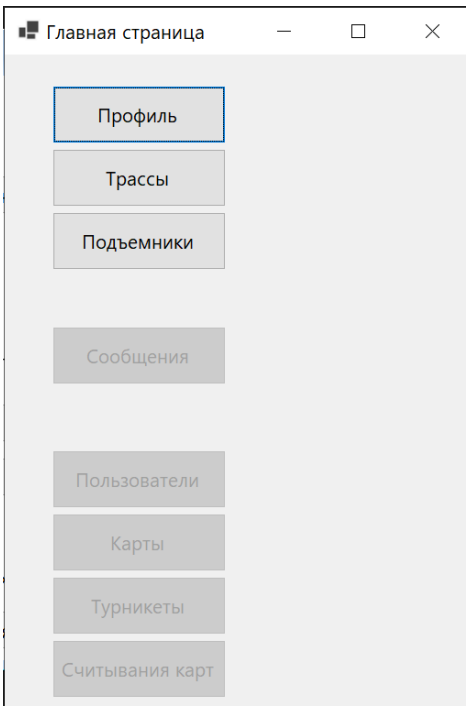


Рис. 3.1: Главное окно приложения (пользователь не авторизован)

В любом окне неактивная кнопка означает, что данное действие недоступно данному пользователю. Например, в окне «Профиль» неавторизованный пользователь может только выполнить вход или зарегистрироваться (рисунок 3.2), авторизованный – только выполнить выход (рисунок 3.3).

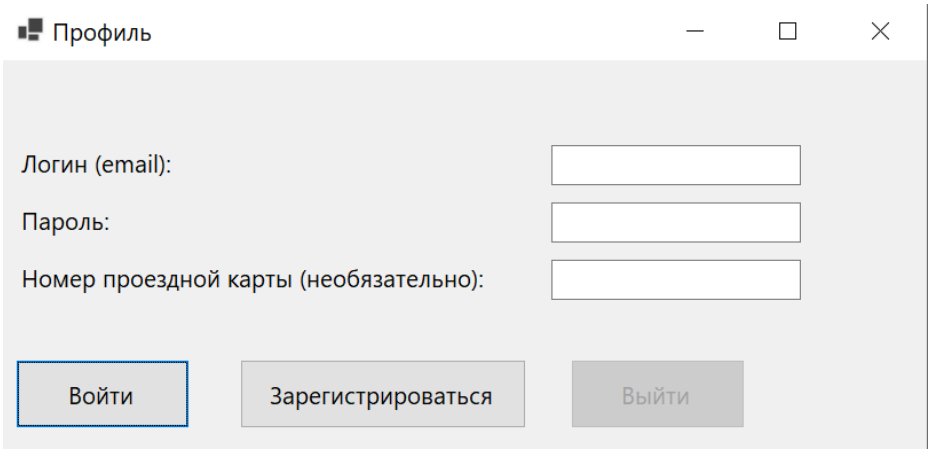


Рис. 3.2: Окно «Профиль» (пользователь не авторизован)

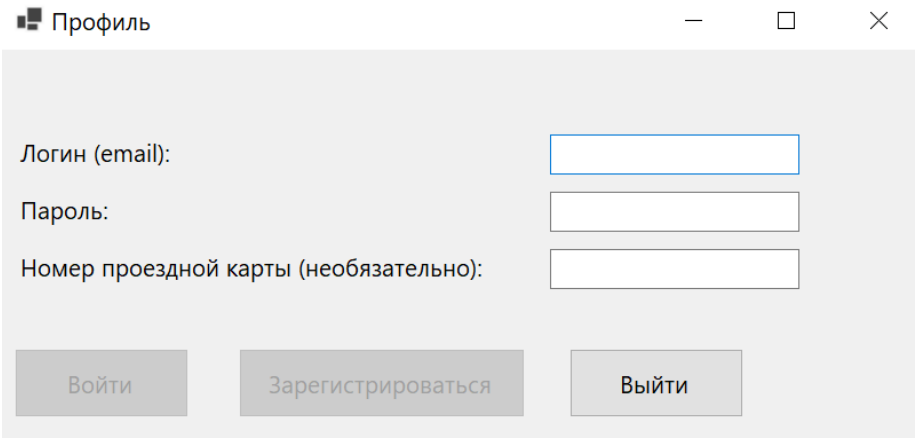


Рис. 3.3: Окно «Профиль» (пользователь авторизован)

Далее для демонстрации всех функций системы окна будут показаны в том виде, в котором они открываются для пользователя с правами администратора, так как ему доступен наибольший функционал.

На рисунке 3.4 показан вид окна «Сообщения». Предоставляется возможность получить информацию обо всех сообщениях или об одном из них по ID; добавить (отправить), изменить (в частности, отметить прочитанным) или удалить сообщение.

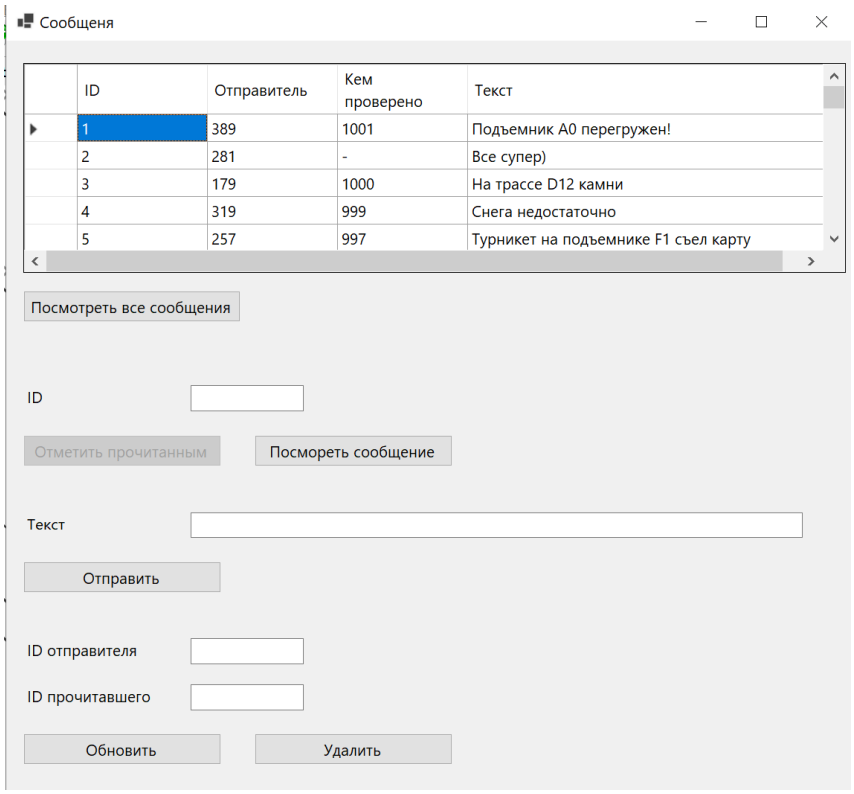


Рис. 3.4: Окно «Сообщения»

На рисунке 3.5 показан вид окна «Трассы». Предоставляется возможность получить информацию обо всех трассах или об одной из них по названию; добавить, изменить или удалить трассу; добавить или удалить связь с подъемником.

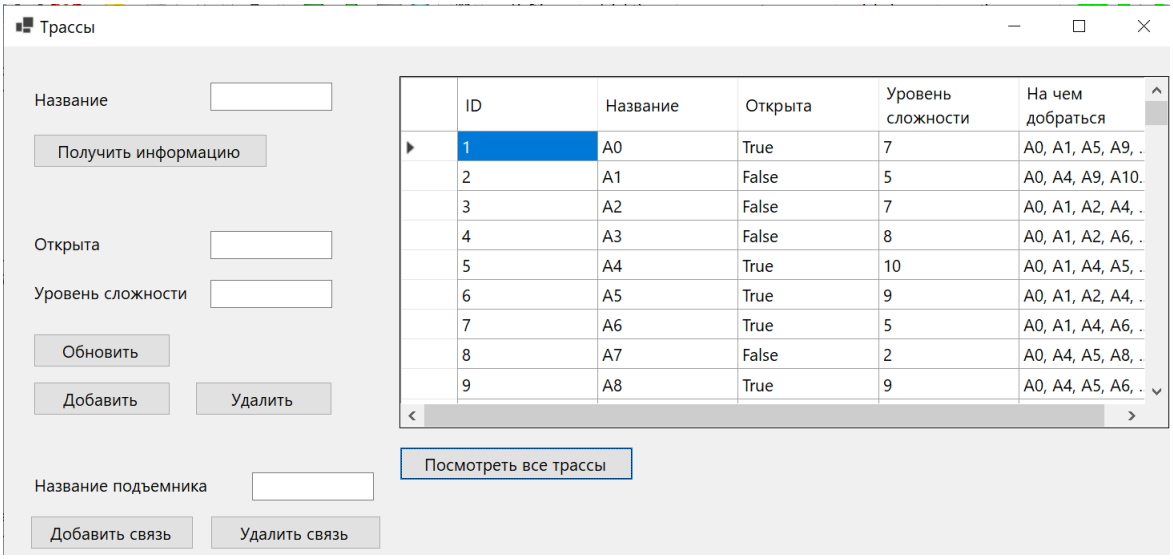


Рис. 3.5: Окно «Трассы»

На рисунке 3.6 показан вид окна «Подъемники». Предоставляется возможность получить информацию обо всех подъемниках или об одном из них по названию; добавить, изменить или удалить подъемник; добавить или удалить связь с трассой.

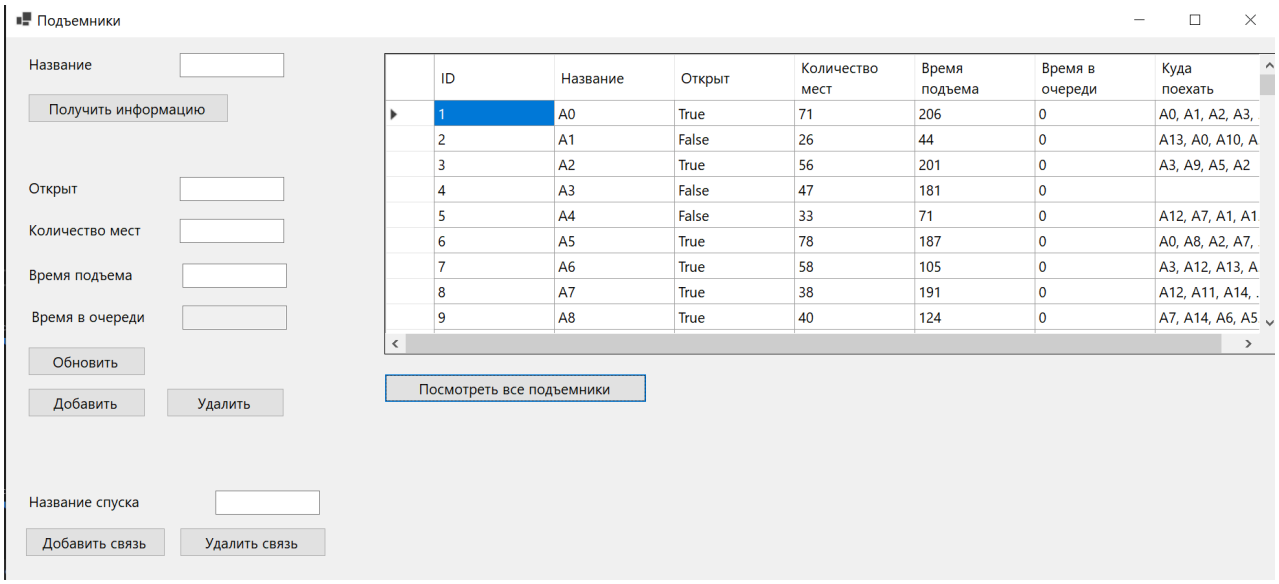


Рис. 3.6: Окно «Подъемники»

На рисунках 3.7, 3.8, 3.9, 3.10 показаны виды окон «Пользователи», «Карты», «Турникеты» и «Считывания карт», соответственно. В каждом окне можно получить информацию обо всех соответствующих объектах или об одном из них по ID; добавить, изменить или удалить объект.

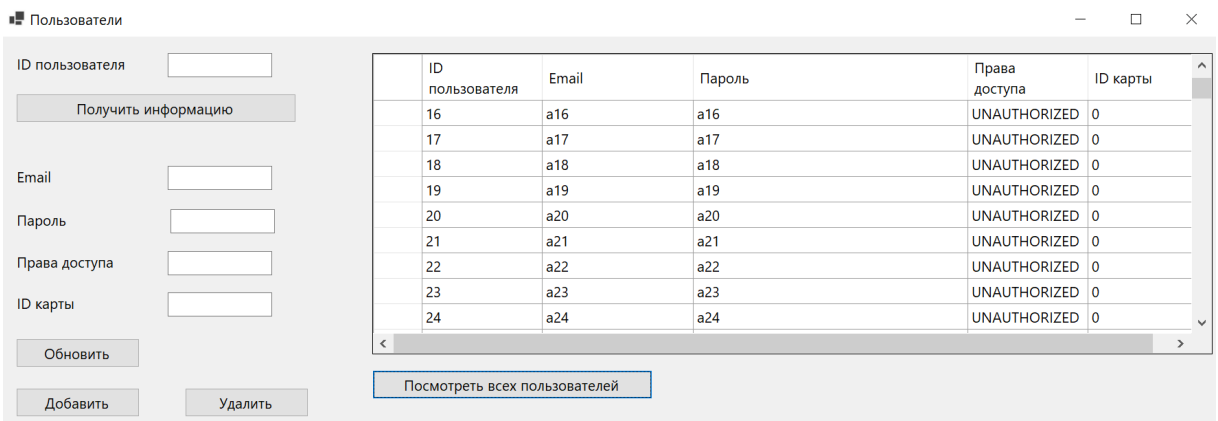


Рис. 3.7: Окно «Пользователи»

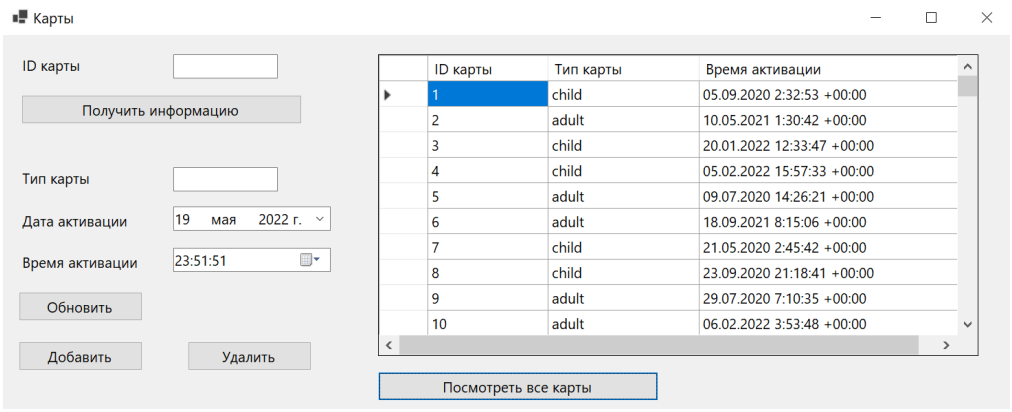


Рис. 3.8: Окно «Карты»

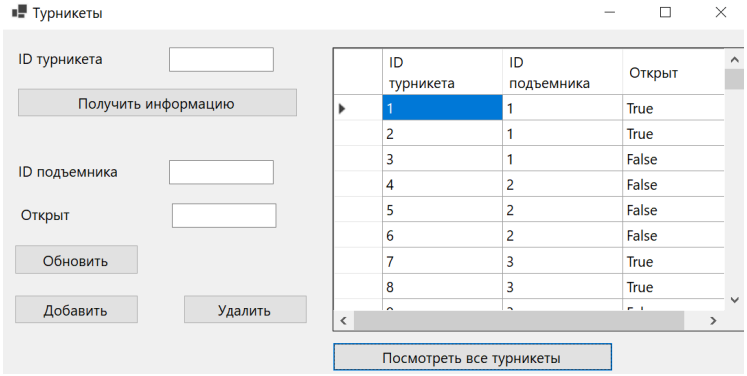


Рис. 3.9: Окно «Турникеты»

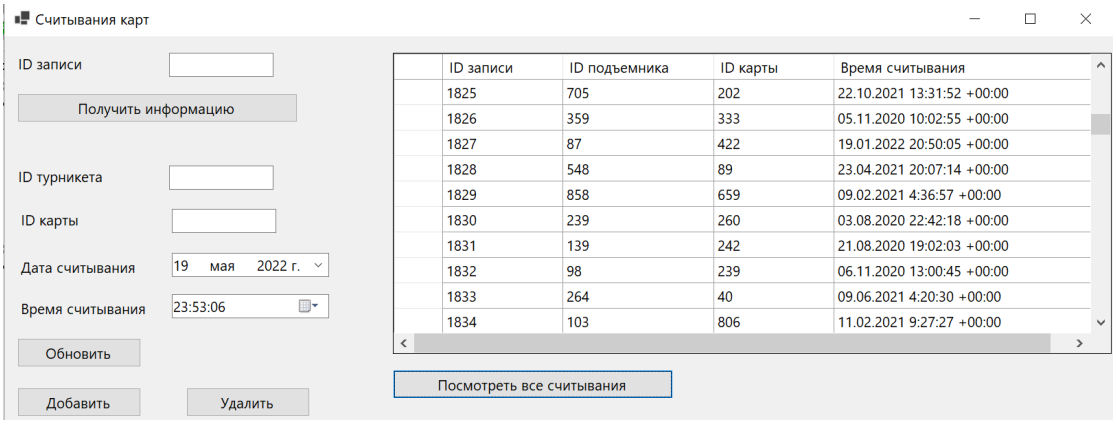


Рис. 3.10: Окно «Считывания карт»

Вывод

На основе анализа in-memory СУБД был осуществлен выбор наиболее подходящей для решения поставленной задачи. Были выбраны инструменты разработки, приведены детали реализации и интерфейс приложения.

4 Исследовательская часть

В данном разделе представлена постановка эксперимента по сравнению времени обновления длительности ожидания в очередях к подъемникам в зависимости от их количества и от того, какой объем вычислений переносится на сторону приложения.

4.1 Цель эксперимента

Целью эксперимента является исследование зависимости времени обновления длительности ожидания в очередях к подъемникам от их количества и от того, где производятся основные шаги вычислений: на стороне БД или на стороне приложения.

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялись измерения:

- операционная система: Windows 10;
- оперативная память: 16 Гб;
- процессор: Intel Core™ i5-8259U.

Во время тестирования ноутбук был включен в сеть питания и нагружен только встроенными приложениями окружения и системой тестирования.

4.3 Описание эксперимента

В эксперименте сравнивается время обновления длительности ожидания в очередях к подъемникам в зависимости от их количества и от используемого алгоритма. Для каждого количества подъемников и каждой

реализации замеры производятся 10 раз, а затем вычисляется среднее время.

4.3.1 Сравниваемые алгоритмы

В эксперименте сравниваются реализации четырех алгоритмов обновления времени ожидания в очередях к подъемникам (описаны в порядке убывания объема вычислений, производимых на стороне БД).

В первом алгоритме бóльшая часть вычислений осуществляется на БД. После получения всех подъемников в цикле для каждого из них вызывается функция update_queue_time (см. листинг 3.2), которая на уровне БД производит все вычисления: поиск связанных с подъемником турникетов, подсчет считываний на этих турникетах в заданный промежуток времени, расчет нового времени ожидания и обновление соответствующего кортежа в спейсе lifts. В листинге 4.1 приведен участок кода, реализующий данный алгоритм.

```
1 List<Lift> lifts =
      _liftsRepository.GetLiftsAsync().GetAwaiter().GetResult();
2 foreach (Lift lift in lifts)
3 {
4   var result = box.Call_1_6<ValueTuple<uint, uint, uint>,
     Int32[]>("update_queue_time", (ValueTuple.Create(lift.LiftID,
     (uint)dateFrom.ToUnixTimeSeconds(),
     (uint)dateTo.ToUnixTimeSeconds()))).GetAwaiter().GetResult();
5 }
```

Листинг 4.1: Первый алгоритм обновления времени ожидания в очередях к подъемникам

Во втором алгоритме после получения всех подъемников в цикле для каждого из них: вызывается функция count_card_readings, которая на уровне БД подсчитывает количество считываний карт в заданный промежуток времени на турникетах, связанных с данным подъемником (см. листинг 3.2); на уровне приложения проводится расчет нового времени ожидания в очереди; осуществляется вызов функции дя обновления соответствующего кортежа в спейсе lifts. В листинге 4.2 приведен участок кода, реализующий данный алгоритм.

```
List<Lift> lifts =
    _liftsRepository.GetLiftsAsync().GetAwaiter().GetResult();
foreach (Lift lift in lifts)
{
  var result = box.Call_1_6<ValueTuple<uint, uint, uint>,
   Int32[]>("count_card_readings",
   (ValueTuple.Create(lift.LiftID,
   (uint)dateFrom.ToUnixTimeSeconds(),
   (uint)dateTo.ToUnixTimeSeconds()))).GetAwaiter().GetResult();
  uint cardReadingsAmout = (uint)result.Data[0][0];

  uint plusQueueTime = cardReadingsAmout * (2 * lift.LiftingTime
   / lift.SeatsAmount);

  uint newQueueTime = Math.Max(lift.QueueTime - timeDelta +
   plusQueueTime, 0);

  Lift updatedLift = new(lift, newQueueTime);
  _liftsRepository.UpdateLiftByIDAsync(updatedLift.LiftID,
   updatedLift.LiftName, updatedLift.IsOpen,
   updatedLift.SeatsAmount,
   updatedLift.LiftingTime).GetAwaiter().GetResult();
}
```

Листинг 4.2: Второй алгоритм обновления времени ожидания в очередях к подъемникам

В третьем алгоритме после получения всех подъемников и считываний карт происходит отбрасывание считываний, которые не попали в указанный промежуток времени, а затем в цикле для каждого подъемника: путем обращения к БД получаются все турникеты, связанные с данным подъемником; на уровне приложения подсчитывается количество считываний карт в заданный промежуток времени на этих турникетах и осуществляется расчет нового времени ожидания в очереди; производится вызов функции дя обновления соответствующего кортежа в спейсе lifts. В листинге 4.3 приведен участок кода, реализующий данный алгоритм.

```
1       List<Lift> lifts =
        _liftsRepository.GetLiftsAsync().GetAwaiter().GetResult();
2   List<CardReading> allCardReadings =
        _cardReadingsRepository.GetCardReadingsAsync()
3   .GetAwaiter().GetResult();
4   List<CardReading> cardReadings = new List<CardReading>();
5   foreach (CardReading cardReading in allCardReadings){
6     if ((cardReading.ReadingTime >= dateFrom) &&
7     (cardReading.ReadingTime < dateTo))
8     cardReadings.Add(cardReading);
9   }
10
11  foreach (Lift lift in lifts){
12    List<Turnstile> turnstiles =
        _turnstilesRepository.GetTurnstilesByLiftIdAsync(lift.LiftID)
13    .GetAwaiter().GetResult();
14    List<uint> connectedTurnstileIDs = new List<uint>();
15    foreach (Turnstile turnstile in turnstiles){
16      connectedTurnstileIDs.Add(turnstile.TurnstileID);
17    }
18
19    uint cardReadingsAmout = 0;
20    foreach (CardReading cardReading in cardReadings){
21      if (connectedTurnstileIDs.Contains(cardReading.TurnstileID))
22      cardReadingsAmout++;
23    }
24    uint plusQueueTime = cardReadingsAmout * (2 * lift.LiftingTime
        / lift.SeatsAmount);
25    uint newQueueTime = Math.Max(lift.QueueTime - timeDelta +
        plusQueueTime, 0);
26
27    Lift updatedLift = new(lift, newQueueTime);
28    _liftsRepository.UpdateLiftByIDAsync(updatedLift.LiftID,
        updatedLift.LiftName, updatedLift.IsOpen,
        updatedLift.SeatsAmount,
        updatedLift.LiftingTime).GetAwaiter().GetResult();
29  }
```

Листинг 4.3: Третий алгоритм обновления времени ожидания в очередях к подъемникам

В четвертом алгоритме бóльшая часть вычислений осуществляется на уровне приложения. Сначала происходит получение всех подъемников, считываний и турникетов из БД. Затем отбрасываются считывания, которые не попали в указанный промежуток времени. Далее в цикле для каждого подъемника: на уровне приложения определяются связанные с ним турникеты; подсчитываются чтения карт на этих турникетах; осуществляется расчет нового времени ожидания в очереди; производится вызов функции дя обновления соответствующего кортежа в спейсе lifts. В листинге 4.4 приведен участок кода, реализующий данный алгоритм.

```
1     List<Lift> lifts =
      _liftsRepository.GetLiftsAsync().GetAwaiter().GetResult();
2  List<Turnstile> turnstiles =
      _turnstilesRepository.GetTurnstilesAsync()
3  .GetAwaiter().GetResult();
4  List<CardReading> allCardReadings =
      _cardReadingsRepository.GetCardReadingsAsync()
5  .GetAwaiter().GetResult();
6  List<CardReading> cardReadings = new List<CardReading>();
7  foreach (CardReading cardReading in allCardReadings)
8  {
9     if ((cardReading.ReadingTime >= dateFrom) &&
10    (cardReading.ReadingTime < dateTo))
11    cardReadings.Add(cardReading);
12 }
13
14 foreach (Lift lift in lifts)
15 {
16    List<uint> connectedTurnstileIDs = new List<uint>();
17    foreach (Turnstile turnstile in turnstiles)
18    {
19       if (turnstile.LiftID == lift.LiftID)
20       connectedTurnstileIDs.Add(turnstile.TurnstileID);
21    }
22
23    uint cardReadingsAmout = 0;
24    foreach (CardReading cardReading in cardReadings)
25    {
26       if (connectedTurnstileIDs.Contains(cardReading.TurnstileID))
27       cardReadingsAmout++;
28    }
```

```
29
30      uint plusQueueTime = cardReadingsAmout * (2 * lift.LiftingTime
        / lift.SeatsAmount);
31      uint newQueueTime = Math.Max(lift.QueueTime - timeDelta +
        plusQueueTime, 0);
32
33      Lift updatedLift = new(lift, newQueueTime);
34      _liftsRepository.UpdateLiftByIDAsync(updatedLift.LiftID,
        updatedLift.LiftName, updatedLift.IsOpen,
        updatedLift.SeatsAmount,
        updatedLift.LiftingTime).GetAwaiter().GetResult();
35  }
```

Листинг 4.4: Четвертый алгоритм обновления времени ожидания в очередях к подъемникам

4.3.2 Используемые данные

В поставленном эксперименте для реализации каждого алгоритма измеряется время обновления длительности ожидания в очередях ко всем подъемникам при их различном количестве: 50, 100, 200, 400, 600, 800. Для каждого подъемника при этом создаются 5 турникетов, для каждого турникета – по 10 считываний, из которых только половина попадает в заданный промежуток времени. Во всех остальных спейсах хранятся 1000 записей в каждом.

4.4 Результаты эксперимента

В таблице 4.1 представлены результаты поставленного эксперимента.

Таблица 4.1: Результаты сравнения времени (мс), необходимого для обновления длительности ожидания в очередях ко всем подъемникам в зависимости от их количества и используемого алгоритма

Количество подъемников \ Номер алгоритма	1	2	3	4
50	891	908	1785	950
100	1760	1797	3593	1822
200	3535	3588	7099	3600
400	7088	7222	14186	7152
600	10657	10754	21363	10676
800	14175	14481	28503	14306

На рисунке 4.1 представлены графики зависимости времени обновления длительности ожидания в очередях к подъемникам от их количества и от используемого алгоритма.

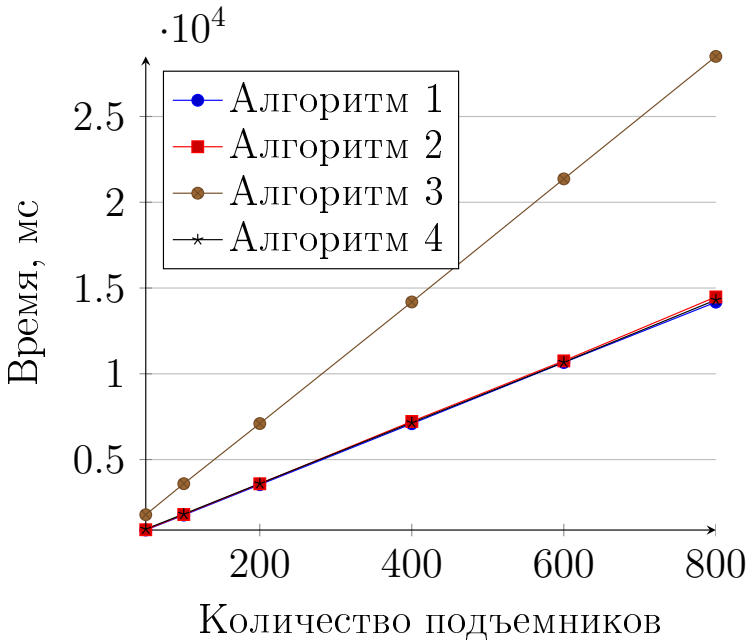


Рис. 4.1: Зависимость времени обновления длительности ожидания в очередях к подъемникам от их количества и от используемого алгоритма

По результатам эксперимента видно, что вне зависимости от используемого алгоритма время, затрачиваемое на обновление длительности ожидания в очередях к подъемникам, линейно зависит от их количества.

Времена, затрачиваемые реализациями первого, второго и четвертого алгоритмов, близки между собой, тогда как реализация третьего алгоритма требует сравнительно больше времени.

При этом наименьшее время было затрачено при использовании первого алгоритма, когда бóльшая часть вычислений осуществляется на БД. В начале к базе данных выполняется одно обращение с получением всех подъемников, а затем для каждого подъемника обращение к ней выполняется только один раз, причем передается небольшой объем информации (3 беззнаковых целых).

Следующая по наименьшему времени обработки – реализация четвертого алгоритма. В данном случае между БД и приложением передается больше данных, чем в первой реализации: в приложении дополнительно единожды запрашиваются все считывания и турникеты из БД, а также происходит передача обновленных данных из приложения обратно в БД для каждого подъемника.

Еще больше времени было затрачено реализацией второго алгоритма. Объем передаваемых данных между приложением и БД увеличивается по сравнению с предыдущими реализациями: в цикле для каждого подъемника выполняется сразу два обращения к БД.

Наконец, наибольшее время использовалось в реализации третьего алгоритма. В этом случае в цикле для каждого подъемника так же, как в предыдущей, выполняется два обращения к БД. Однако в реализации второго алгоритма помимо передачи обновленных данных из приложения в БД для каждого подъемника требовалось получить одно число – количество считываний. В реализации же третьего алгоритма второе обращение к БД выполняется для получения бóльшего объема данных: все связанные с подъемником турникеты – это сразу несколько кортежей.

Вывод

В результате поставленного эксперимента была выявлена линейная зависимость времени обновления длительности ожидания в очередях к подъемникам от их количества. Исследование также показало, что при фиксированном количестве подъемников на время выполнения обновления бóльшее влияние оказывает не то, где производятся основные шаги вычислений (на стороне БД или на стороне приложения), а объем данных, передаваемых между приложением и БД.

Заключение

Цель курсовой работы достигнута, все поставленные задачи решены. Была разработана база данных для онлайн-мониторинга состояния трасс и подъемников горнолыжного курорта.

В ходе выполнения работы были изучены способы хранения данных, рассмотрены существующие СУБД, получены знания в области проектирования БД и приложений. Также было реализовано программное обеспечение, позволяющее получать доступ к данным.

В ходе выполнения экспериментально-исследовательской части было установлено, что на время выполнения операций, требующих обращения к БД бóльшее влияние оказывает не то, где производятся основные шаги вычислений (на стороне БД или на стороне приложения), а объем данных, передаваемых между приложением и БД.

Список использованных источников

[1] Интерес туристов из РФ к горнолыжному отдыху [Электронный ресурс]. Режим доступа: https://booking-ru.ru/2020/09/20/interes-turistov-iz-rf-k-gornolyzhnomu-otdyxu-vyros-na-tret/ (дата обращения: 01.04.2022).

[2] What is a REST API? - Red Hat [Электронный ресурс]. Режим доступа: https://www.redhat.com/en/topics/api/what-is-a-rest-api (дата обращения: 01.04.2022).

[3] Что такое база данных | Oracle Россия и СНГ [Электронный ресурс]. Режим доступа: https://www.oracle.com/ru/database/what-is-database/ (дата обращения: 25.03.2022).

[4] Системы и технологии баз данных в памяти | Microsoft [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/sql/relational-databases/in-memory-database?view=sql-server-ver15 (дата обращения: 25.03.2022).

[5] In-Memory. База данных в оперативной памяти |ECM-Journal [Электронный ресурс]. Режим доступа: https://ecm-journal.ru/material/In-Memory-Baza-dannykh-v-operativnojj-pamjati (дата обращения: 27.03.2022).

[6] In-Memory Database [Электронный ресурс]. Режим доступа: https://ru.bmstu.wiki/IMDB_(In-memory_Database) (дата обращения: 27.03.2022).

[7] 4 крупных примера внедрения Tarantool | Big Data School [Электронный ресурс]. Режим доступа: https://www.bigdataschool.ru/blog/tarantool-use-cases-and-advantages.html (дата обращения: 27.03.2022).

[8] Memcached - a distributed memory object caching system [Электронный ресурс]. Режим доступа: https://memcached.org/ (дата обращения: 15.04.2022).

[9] Memcached | Распределенное хранилище данных типа «ключ-значение» [Электронный ресурс]. Режим доступа: https://memcached.org/ (дата обращения: 15.04.2022).

[10] Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker [Электронный ресурс]. Режим доступа: https://redis.io/ (дата обращения: 01.04.2022).

[11] Преимущества Redis [Электронный ресурс]. Режим доступа: https://aws.amazon.com/ru/redis/ (дата обращения: 01.04.2022).

[12] ACID (Atomicity, Consistency, Isolation, Durability) [Электронный ресурс]. Режим доступа: https://ru.bmstu.wiki/ACID_(Atomicity,_Consistency,_Isolation,_Durability) (дата обращения: 01.04.2022).

[13] Tarantool – Платформа In-memory вычислений [Электронный ресурс]. Режим доступа: https://www.tarantool.io/ru/ (дата обращения: 01.04.2022).

[14] The Programming Language Lua [Электронный ресурс]. Режим доступа: http://www.lua.org/ (дата обращения: 01.04.2022).

[15] Документация C# [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения: 03.04.2022).

[16] Dotnet client for Tarantool NoSql database [Электронный ресурс]. Режим доступа: https://github.com/progaudi/progaudi.tarantool (дата обращения: 03.04.2022).

[17] Документация по семейству продуктов Visual Studio [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/?view=vs-2022 (дата обращения: 03.04.2022).

[18] Windows Forms [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/desktop/winforms/?view=netdesktop-5.0 (дата обращения: 03.04.2022).

[19] Docker: Empowering App Development for Developers [Электронный ресурс]. Режим доступа: https://www.docker.com/ (дата обращения: 03.04.2022).

[20] Documentation site for the xUnit.net unit testing framework [Электронный ресурс]. Режим доступа: https://xunit.net/ (дата обращения: 03.04.2022).