МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет вычислительной техники

Кафедра «Вычислительная техника»

Направление подготовки 09.04.01 «Информатика и вычислительная техника»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

ИССЛЕДОВАНИЕ РЕКОМЕНДАТЕЛЬНЫХ СИСТЕМ НА ОСНОВЕ ГРАФОВЫХ МОДЕЛЕЙ

Студент 20.06.2024 (подпись, дата) Гришин Евгений Валентинович (ФИО полностью)

Руководитель 20.06.2024 (подпись, дата) Л.В. Гурьянов (фамилия, инициалы)

Нормоконтролёр 20.06.2024 (подпись, дата) А.С. Бычков (фамилия, инициалы)

Рецензент (подпись, дата) А.В. Васильев (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от 27.05.2024 № 11)

Заведующий кафедрой (подпись) М.А. Митрохин (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 21.06.24 № 37)

Секретарь ГЭК (подпись) Е.И. Гурин (фамилия, инициалы)

Пенза 2024

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет вычислительной техники

Кафедра «Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
_____М.А. Митрохин
«13» сентября 2022 г

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

1. Студент Гришин Евгений Валентинович гр. 22ВВИм2 факультета ВТ

2. Тема работы «Исследование рекомендательных систем на основе графовых моделей»

Тема утверждена приказом ПГУ № 765/о от « 13 » сентября 2022 г.

3. Руководитель работы к.т.н., доцент кафедры «МОиПЭВМ» ПГУ Гурьянов Л.В.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Назначение: рекомендательная система для выбора наиболее интересных элементов для просмотра пользователем, ограниченная в вычислительных и пространственных ресурсах.

2) Среда выполнения: операционные системы Windows 10, Ubuntu.

Языковые средства разработки: Java, Kotlin, UML.

Инструментальные средства: IntelliJ IDEA Community Edition,

Технологии разработки: ООП, функциональное программирование, Kotlin Coroutines, сериализация объектов, модульное тестирование.

3) Основные функции программных средств:

а) выбор элементов-кандидатов для рекомендации в прикладных приложениях;

б) индексирование взаимодействий пользователей с элементами в модели;

в) распространение проиндексированных моделей между серверами, предоставляющими рекомендации элементов;

г) ограничение вычислительных и пространственных ресурсов для алгоритмов;

д) оптимизация множественных взаимодействий между пользователем и элементом;

е) совместимость с серверными приложениями, реализованными на языке Java.

5. Перечень подлежащих разработке вопросов

1) *анализ предметной области и требований;*
2) *проектирование;*
3) *реализация;*
4) *оптимизация программного решения;*
5) *тестирование;*
6) *оформление пояснительной записки.*

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований	10%	13.09.2022 – 15.11.2022	
2	Проектирование	20%	16.11.2022 – 17.03.2023	
3	Реализация	40%	18.03.2023 – 19.11.2023	
4	Оптимизация программного решения	10%	20.11.2023 – 19.01.2024	
5	Тестирование	10%	20.01.2024 – 19.03.2024	
6	Оформление пояснительной записки	10%	20.03.2024 – 21.05.2024	

Дата выдачи задания «13» __сентября_____2022 г.

Руководитель бакалаврской работы _____ Гурьянов Л.В.
(подпись, дата)

Задание к исполнению принял студент _____ Гришин Е.В.
(подпись, дата)

Работу к защите допустить

Декан факультета 20.06.2024 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Пояснительная записка 81 с., 25 рис., 4 табл., 14 источн., 3 прил.

РЕКОМЕНДАТЕЛЬНЫЕ СИСТЕМЫ, ГРАФЫ, СТОХАСТИЧЕСКИЕ ОБХОДЫ, КОТЛИН, ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ, СЕРИАЛИЗАЦИЯ

Объект исследования – рекомендательные системы на базе графовых моделей предметной области.

Цель работы – разработка рекомендательной системы, способной функционировать без распределения пользователей по кластерам и в условиях ограниченных вычислительных и пространственных ресурсов.

Технологии разработки – разработка библиотеки проводилась на языке программирования Kotlin и является полностью совместимой с серверными приложениями, реализованными на языке Java. В качестве среды разработки использовалась интегрированная среда разработки IntelliJ Idea Community Edition.

Среда исполнения – Windows 10, Ubuntu.

Результаты работы – в ходе выполнения выпускной квалификационной работы была разработана библиотека, позволяющая интегрировать в серверное приложение, исполняемое на JVM, функционал рекомендательной системы. Все требования к разработке приложения соблюдены. Проведенное модульное тестирование библиотеки показало корректность работы ее компонентов.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ 4
ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 5
ВВЕДЕНИЕ 6
1 Анализ рекомендательных систем 8
1.1 Теоретическая основа рекомендательных систем 8
1.2 Сравнительный анализ алгоритмов рекомендательных систем 13
1.3 Анализ требований к разработке 17
1.3.1 Назначение разработки 17
1.3.2 Требования к программной реализации 17
1.4 Концепция разрабатываемой рекомендательной системы 19
2 Разработка графовой рекомендательной системы 25
2.1 Графовые модели рекомендательной системы 25
2.2 Организация структуры хранения данных графовых моделей 27
2.3 Процесс индексирования модели рекомендательной системы 30
2.4 Подготовка моделей системы к горизонтальному масштабированию 32
2.5 Предлагаемый алгоритм анализа элементов 34
2.6 Оптимизация скорости исполнения запросов 36
3 Оценка результатов исследования 38
3.1 Тестирование совместимости 38
3.2 Тестирование корректности 39
3.3 Тестирование эффективности 44
3.4 Методология маркетингового исследования при внедрении 45
3.5 Дальнейшие перспективы исследования 49
ЗАКЛЮЧЕНИЕ 50
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 51
ПРИЛОЖЕНИЕ А. Иллюстрационные материалы 53
ПРИЛОЖЕНИЕ Б. Листинг 56
ПРИЛОЖЕНИЕ В. Презентация 75

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Термин	Определение
Рекомендательная система	Программное обеспечение, осуществляющее подбор наиболее интересных для пользователя объектов предметной области на основе его предпочтений, сбор данных о взаимодействии пользователя с системой, а также адаптацию алгоритмов под изменяющийся профиль предпочтений пользователя
Рекомендательный движок	Компонент рекомендательной системы, реализующий алгоритмы для фильтрации и персонализации наиболее релевантных объектов предметной области
Релевантность	Степень соответствия найденного объекта нуждам пользователя
Пользователь	Субъект, обращающийся к информационной системе для получения необходимой ему информации
Рекомендация (рекомендуемый элемент)	Объекты информационной системы, которые оцениваются рекомендательной системой по релевантности и предоставляются конечному пользователю
Агрегатор элементов	Объекты или субъекты информационной системы, группирующие рекомендуемые элементы
Кандидат для рекомендации	Объекты информационной системы, прошедшие первоначальную фильтрацию рекомендательными алгоритмами, но не персонализированные под профиль предпочтений пользователя
Индексирование	Процесс формирования рекомендательной системой модели данных на основе собранных данных об объектах и субъектах информационной системы
Сериализация	Процесс перевода модели данных в формат, доступный для хранения и передачи
Десериализация	Процесс перевода сериализованных данных в структуру для дальнейшей их обработки
Стохастический алгоритм	Алгоритм, поведение которого зависит от случайных факторов

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей работе применяются следующие сокращения и обозначения

ООП – объектно-ориентированное программирование

СУБД – система управления базами данных

CBOR – Concise Binary Object Representation, краткое представление двоичных объектов

CSV – Comma-Separated Values, разделенные запятой значения

CTR – Click-Through Rate, отношение кликабельности объекта

GPU – графическое процессорное устройство

JSON – JavaScript Object Notation, нотация объектов языка JavaScript

JVM – Java Virtual Machine, виртуальная машина Java

KPI – Key Performance Indicator, ключевой показатель эффективности

O – Нотация большой «O», обозначает верхний предел асимптотики исполнения алгоритма

ProtoBuf – Protocol Buffers, технология бинарной сериализации Google

RNSA – Refined Neighbor Search Algorithm, улучшенный алгоритм поиска соседей

SALSA – Stochastic Approach for Linked Structure Analysis, стохастический подход к анализу связных структур

ВВЕДЕНИЕ

Рекомендательные системы являются важнейшей составляющей крупных веб-порталов, будь то социальная сеть, видеохостинг или поисковой сервис. Именно с их помощью пользователи получают возможность просматривать ту информацию, которая отличается от страниц, которые они посещают обычно, но при этом является похожей, делая сервисы удобнее и разнообразнее. Для того, чтобы предложить пользователю интересные ему статьи, видео или фото, применяются различные алгоритмы, различающиеся скоростью, используемой информацией о рекомендуемых сущностях, а также проблемами, решение которых стоит перед программистом. Среди проблем рекомендательных систем можно выделить: проблему холодного старта пользователя, проблему холодного старта контента, проблему разнообразия рекомендаций, проблему обновления рекомендаций и т.д. Различные алгоритмы имеют индивидуальные преимущества и недостатки в решении этих проблем.

Одной из компаний, которая предложила использующее графовые модели решение, стала Pinterest. Их рекомендательная система Pixie [1] используется для обработки терабайтов данных и быстрого формирования рекомендаций. Кроме того, подобная рекомендательная система на графовых моделей GraphJet [2] используется в экосистеме проекта Twitter для формирования персонализированных рекомендаций твитов и пользователей платформы микроблоггинга. Простота и эффективность этого подхода заключается в использовании графового представления предметной области.

Целью исследования является разработка рекомендательной системы, способной функционировать на единственном устройстве и предоставлять рекомендации в режиме реального времени.

При разработке были модифицированы уже существующие концепции и решения, что привело к системе, совмещающей их сильные стороны и удовлетворяющей требованиям. Библиотека графовой рекомендательной

системы может использовать различные виды моделей: невзвешенные простые графы, взвешенные простые графы и взвешенные мульти-графы.

Для распространения моделей используется сериализация в бинарный формат CBOR. С помощью редакторов, реализованных в составе библиотеки, возможно индексирование данных предметной области и конвертирование графовых моделей между форматами хранения.

Для генерации кандидатов для рекомендации используются несколько видов запросов, которые можно адаптировать под различные виды систем: поиск элементов, похожих на выбранный элемент, поиск элементов, похожих на набор элементов (удобно для генерации рекомендаций, похожих на историю просмотра пользователя), поиск элементов, связанных с агрегатором (например, пользователем прикладной системы), и поиск элементов, связанных с множеством агрегаторов (например, пользователями сообщества или друзьями). Гибкая природа разработки позволяет ей адаптироваться к условиям различных прикладных программных средств.

1 Анализ рекомендательных систем

1.1 Теоретическая основа рекомендательных систем

Рекомендательные системы являются прикладными программными продуктами, улучшающими эффективность приложений, в которых пользователи взаимодействуют с элементами предметной области. Грамотная интеграция решения позволяет увеличить бизнес-эффективность за счет увеличения числа положительных взаимодействий клиентов с сервисом. Наиболее популярными примерами сервисов, которые получают из этого выгоду, являются: интернет-магазины, социальные сети, видеохостинги и интерактивные образовательные курсы. Утилизируя решения в области рекомендательных систем, приложения повышают прибыль, предлагая интересные продукты, уменьшают время, которое пользователь проводит в поиске интересной ему информации, или улучшают качество системы, адаптируя систему под персональные нужды пользователя.

Проектирование рекомендательных систем является сложным процессом, который требует принятия решений в отношении ряда проблем. Наиболее сложными являются проблемы холодного старта пользователей и элементов сервиса. Новые объекты, добавленные в систему, не содержат информации, требуемой для персонализации рекомендаций. Холодные элементы системы трудно различимы и не имеют взаимодействий с пользователями, из-за чего они не могут попасть в рекомендации пользователей с выстроенным профилем предпочтений. Холодные пользователи изначально не имеют взаимодействий с сервисом, что не позволяет сформировать их персонализированные рекомендации. Для решения этих проблем, приложения часто собирают информацию вне системы или продвигают новые объекты системы для увеличения взаимодействий с ними [3].

Другой проблемой рекомендательных систем является поддержания разнообразия рекомендаций. Если предлагать пользователю одни и те же элементы, он не будет исследовать другие уголки системы. Для борьбы с этим, адаптируются стохастические алгоритмы, которые, кроме похожих элементов, случайным образом добавляют другие объекты сервиса к выборке.

Так как сервисы могут быть огромными по охвату пользователей и контента, с проектированием рекомендательной системы ассоциируется и высокая вычислительная нагрузка. Часто, при постановке задачи рекомендательной системы используют матрицу предпочтений из N столбцов и M строк. На пересечении столбца и строки находится рейтинг элемента для конкретного пользователя. Следовательно, только на хранение такой матрицы асимптотически уходит $O(NM)$ памяти, где N – количество пользователей, а M – количество рекомендуемых элементов. Частично эта проблема решается использованием разреженных матриц, однако к некоторым из классических алгоритмов, этот подход неприменим [3].

Проблема масштабируемости часто решается попытками кластеризации пользователей и рекомендуемых элементов. Используя эту технику, системы уменьшают объем информации, используемый при формировании профиля пользователя или свойств рекомендуемых элементов, что приводит к уменьшению пространственных и вычислительных ресурсов, однако изолирует рекомендации от сущностей вне кластера. Так, пользователи при распределении в кластер, никогда не увидят в рекомендациях элементы, предлагаемые для пользователей в других кластерах.

Для того, чтобы адаптироваться к таким условиям, часто используется архитектура рекомендательных систем, представляющая собой конвейер из трех этапов:

– генераторы кандидатов – алгоритмы, способные выделить из большого количества объектов сервиса разнообразные и релевантные для интересующего субъекта,

– ранкеры – алгоритмы, оценивающие элементы системы с точки зрения персонализации для пользователя. За счет снижения количества элементов на этапе выбора кандидатов, могут быть более вычислительно затратными и иметь более высокую точность. На этом этапе часто прогнозируется рейтинг рекомендуемого элемента для интересующего субъекта,

– миксеры – компоненты, осуществляющие фильтрацию рекомендаций и перемешивание элементов из различных источников рекомендаций. На этом этапе могут быть добавлены коммерчески продвигаемые элементы, а также холодные объекты.

Наиболее явно такая архитектура прослеживается на примере рекомендательной системы Twitter, которая была опубликована в процессе исследования (см. рисунок А.1). Различные алгоритмы генерации кандидатов для рекомендации, использующие нейросетевые, графовые, вероятностные и эвристические подходы, применяются в миксерах для поиска элементов, генерации основной ленты, предложения пользователей для просмотра, после чего персонализируются нейросетевым ранкером под конкретного пользователя. Среди наиболее высоко оцененных элементов отфильтровываются те, которые были просмотрены пользователем или отмечены как игнорируемые. После этого добавляются рекламируемые элементы, корректируется баланс новизны контента и формируется итоговая рекомендательная выборка.

Рассмотрим классификацию и основные подходы к проектированию рекомендательных систем. Чаще всего алгоритмы рекомендательных систем разделяют на две группы (см. рисунок 1).

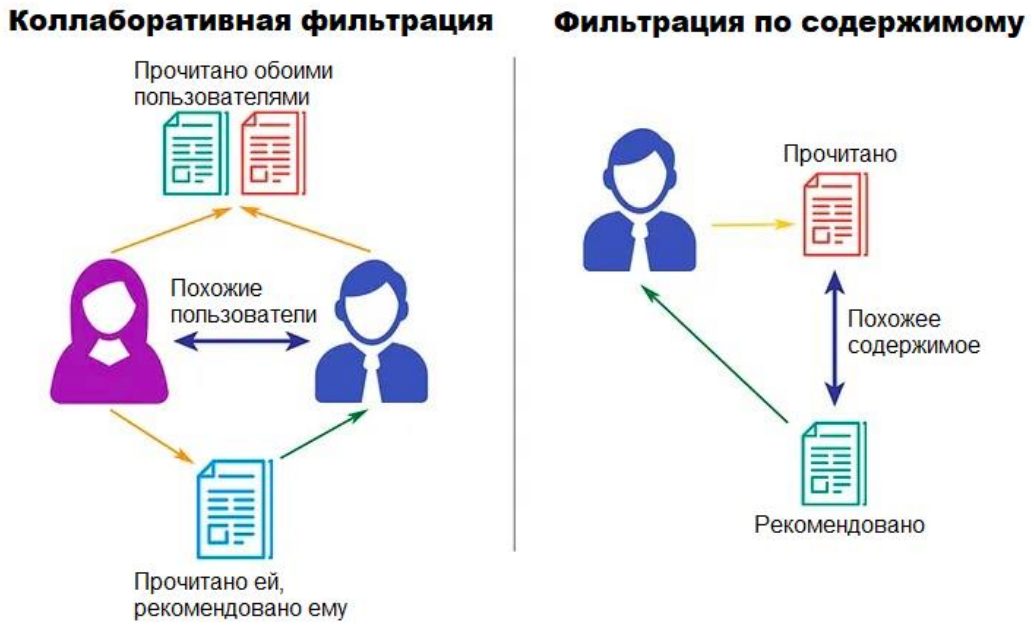


Рисунок 1 – Принцип работы алгоритмов коллаборативной фильтрации и на основе содержимого

Алгоритмы коллаборативной фильтрации используют метаданные об агрегаторах, пользователях и их взаимодействиях для сопоставления похожих субъектов рекомендательной системы и выбора элементов, которые просмотрены или высоко оценены похожими субъектами, но не были предложены интересующему. В этой категории часто используются классические для рекомендательных систем математические модели, такие как [4]:

– сингулярное разложение,

– Байесовские классификаторы,

– использование доверительных интервалов,

– RNSA (усовершенствованный алгоритм выбора соседей) утилизирует коэффициенты корреляции Пирсона и кластеризацию методом К-средних.

Использование этих моделей в комбинации с другими рекомендательными алгоритмами, может помочь при поиске решения для некоторых задач, стоящих при проектировании рекомендательной системы. Формирование рейтинга контента на основе доверительных интервалов осуществляет поддержку в борьбе с проблемой холодного старта пользователя, ведь рейтинг контента не зависит от наличия пользователя в системе. Сингулярное разложение великолепно показало себя в рекомендательных системах, так как часто данные являются разреженными, что указывает на то, что значительная часть содержимого не была оценена пользователями.

Алгоритмы на основе содержимого пытаются сравнивать рекомендуемые сущности и подбирать для пользователя статьи/видео/музыку похожего содержимого. Такие алгоритмы сильно зависимы от качества модели, полученной после долгого процесса обучения.

Вариантов реализации подобных моделей несколько:

– обучение классификатора элементов системы и генерация профиля пользователя по элементам, просмотренным в прошлом, что позволяет предсказать, как этот контент будет воспринят. Пример: возьмем музыкальный сервис и классифицируем жанры треков, которые на нем встречаются. Теперь

пользователю, который уже прослушал какие-то композиции, можно предлагать контент такого же жанра и похожих исполнителей [5],

– Сформируем набор скрытых признаков, которые будут генерироваться нашей моделью машинного обучения. Для того контента, который пользователю понравился, мера расстояния между скрытыми признаками пользователя и рекомендуемыми элементами, должна быть минимизирована. В случае отрицательных взаимодействий, мера расстояния должна быть максимизирована. Здесь часто используются огромные наборы данных и глубокие нейронные сети для обработки естественного языка или изображений.

Скрытые признаки могут быть представлены скрытыми факторами (Latent Factors) или обучаемыми в нейронных сетях векторными представлениями (Embeddings). Скрытые признаки представляют собой предпочтения конкретного пользователя в виде численных значений, которые мы можем вычислить с помощью модели, лишь основываясь на значениях наблюдаемых факторов, таких как рейтинг [3].

Несмотря на то, что алгоритмы рекомендаций, основанные на содержимом, намного тяжелее в подготовке и дороже в обслуживании, у них есть положительные стороны. При использовании таких рекомендаций практически отсутствует проблема холодного старта контента, так как пользователям не нужно найти новый объект и оценить его для того, чтобы он был добавлен в модель. Достаточно запланированной при добавлении элемента процедуры подготовки с предподсчетом его скрытых факторов.

В случае холодного старта пользователя, новые клиенты в течение некоторого времени не могут получать рекомендации. С рекомендуемыми элементами, некоторые алгоритмы требуют определенного количества взаимодействий с ними до состояния, в котором они становятся активными.

Такая классификация подчеркивает необходимость использования набора алгоритмов в конечном продукте, поскольку различные подходы имеют свои преимущества и недостатки. В случае совместной фильтрации выражена проблема холодного старта рекомендуемых элементов, так как пользователи с

ними не взаимодействовали. В случае фильтрации по содержимому выражена проблема холодного старта пользователя, так как элементы сравниваются между собой при появлении, а для новых пользователей не рассчитан профиль предпочтений.

1.2 Сравнительный анализ алгоритмов рекомендательных систем

Рассмотрим четыре основных класса алгоритмов, используемых при проектировании рекомендательных систем:

1) статистические алгоритмы рекомендательных систем,
2) применение алгоритмов классического машинного обучения,
3) графовые подходы к анализу рекомендательных систем,
4) архитектуры, утилизирующие глубокое обучение нейронных сетей.

Модели рекомендательных систем восходят к методам построения экспертных систем, и одним из первых способов являлось использование ассоциативных правил. Однако этот подход отличается высоким вмешательством эксперта в процесс, низкой динамичностью рекомендаций и медленным циклом обновления системы при добавлении нового контента. Поэтому, ему на смену пришли алгоритмы, которые на текущий момент разделяют на два класса: коллаборативная фильтрация и фильтрация по содержимому. Алгоритмы классического машинного обучения, а также вероятностные модели были рассмотрены в предыдущем подразделе.

В случае применения коллаборативной фильтрации, алгоритмы направлены на корректное сопоставление пользователей системы по их действиям в системе. В результате, пользователи, чья активность наиболее коррелирует относительно некоторой метрики, получают похожие элементы для просмотра [6]. Противопоставлены им алгоритмы, сравнивающие содержимое рекомендуемого контента. Большой проблемой является сложность анализа различных типов данных. Однако, с расширением применения нейронных сетей в областях компьютерного зрения и естественного языка, появились новые возможности для построения таких систем. На текущий момент нейронные сети используются для

оценки схожести двух рекомендуемых элементов или составлению векторных представлений (эмбеддингов) [7] содержимого по данным их использования.

Для проведения сравнительного анализа, в качестве критериев выбраны наиболее важные возможности, которые напрямую влияют на эффективность разработки и использования рекомендательной системы, а именно:

– функциональные возможности:

1) вычислительная сложность (ниже сложность – выше оценка);

2) используемый объем пространства (ниже объем – выше оценка);

3) релевантность рекомендаций (выше релевантность – выше оценка);

4) разнообразие рекомендаций (выше разнообразие – выше оценка);

5) изолированность элементов (ниже изолированность – выше оценка);

6) уязвимость к холодному старту (ниже уязвимость – выше оценка);

– нефункциональные возможности:

1) доступность реализации (выше доступность – выше оценка);

2) распространенность применения (выше популярность – выше оценка);

– технические возможности:

1) необходимость графического ускорителя (ниже необходимость – выше оценка);

2) сложность реализации на JVM-сервере (ниже сложность – выше оценка).

В результате исследования различных алгоритмов и моделей рекомендательных систем была сформирована оценка коэффициентов значимости критериев, которые представлены в таблице 1.

Коэффициенты значимости варьируются от 0 до 1. При значении коэффициента 0, соответствующий критерий не влияет на использование системы и может быть проигнорирован. При значениях коэффициента, близких к 1, критерий является важным качеством системы, на который следует обратить пристальное внимание.

Таблица 1 – Критерии оценки алгоритмов рекомендательных систем

Наименование	Коэффициент значимости
Функциональные возможности	
Вычислительная сложность	0,8
Используемый объем пространства	1
Релевантность рекомендаций	0,7
Разнообразие рекомендаций	0,6
Изолированность элементов	0,8
Уязвимость к холодному старту	0,5
Нефункциональные возможности	
Доступность реализации	0,5
Распространенность применения	0,25
Технические возможности	
Необходимость GPU	0,9
Сложность реализации на JVM-сервере	0,8

Для оценки описанных выше критериев используется пятибалльная шкала соответствия, от 0 – «полное несоответствие», до 4 – «полное соответствие» требованиям.

Таблица 2 заполнена на основе оценки описанных выше алгоритмов. Расчет аддитивной суммы интегральной оценки для каждого сравниваемого продукта проведен по следующей формуле:

$$О_{пп} = \sum_{i=1}^{n}(Z_i * K_i),$$

где $О_{пп}$ – интегральная оценка программного продукта;

n – количество критериев сравнения заявленных продуктов согласно таблице критериев;

Z_i – значение выполнения критерия (4 – выполнен, 0 – не выполнен);

K_i – коэффициент значимости критерия сравнения (от 0 до 1).

Таблица 2 – Критериальная оценка алгоритмов рекомендательных систем

№	Критерий	Алгоритм №			
		1	2	3	4
1	Вычислительная сложность	4	1	3	2
2	Используемый объем пространства	4	1	3	1
3	Релевантность рекомендаций	2	3	3	4
4	Разнообразие рекомендаций	1	2	4	2
5	Изолированность элементов	2	0	4	3
6	Уязвимость к холодному старту	4	1	2	3
7	Доступность реализации	4	2	3	1
8	Распространенность применения	3	3	3	2
9	Необходимость GPU	4	3	4	0
10	Сложность реализации на JVM-сервере	3	2	3	1
	Интегральная оценка	21,55	11,65	22,35	12,3

В результате приходим к выводу, что наиболее подходящими для решения задач, стоящих перед разрабатываемой рекомендательной системой, являются статистические и графовые алгоритмы. Статистические алгоритмы отличаются популярностью, доступностью и устойчивостью к проблеме холодного старта. Графовые рекомендательные движки предлагают разнообразие рекомендаций при контролируемых вычислительных и пространственных ресурсах. Нейросетевые алгоритмы неудобны для решения текущих задач, так как требуют использования графического ускорителя для быстрого обучения и предоставления рекомендаций, а классические алгоритмы машинного обучения утилизируют кластеризацию, способствующую изолированности элементов.

Итоги анализа аналогов указывают на то, что разработка с использованием графовых моделей актуальна, так как эта реализация обладает преимуществами в условиях поставленных требований.

1.3 Анализ требований к разработке

1.3.1 Назначение разработки

Разрабатываемое программное обеспечение должно представлять собой библиотеку, которая при встраивании в приложение организует механизмы рекомендательной системы с совместной фильтрацией.

Функционал библиотеки используется на следующих этапах работы приложения:

– индексирование и сериализация модели на основе данных о рекомендуемых элементах,

– десериализация и подготовка модели к использованию,

– выбор элементов-кандидатов для рекомендации.

Сущности-агрегаторы и рекомендуемые элементы должны иметь уникальные идентификаторы, которые используются в модели и способны быть использованы для однозначного декодирования в приложении.

Программное обеспечение должно быть совместимо с языками, работающими с Java Virtual Machine (JVM), такими как Java, Kotlin или Scala.

1.3.2 Требования к программной реализации

Библиотека должна обеспечивать возможность выполнения перечисленных ниже функций:

– индексирование зависимостей (в том числе многократных и взвешенных) между элементами и агрегаторами произвольным образом (зависимости коллекция-элемент, пользователь-элемент),

– сериализация моделей для хранения и последующего распространения между рекомендательными сервисами,

– десериализация загруженных моделей,

– генерация персонализированных рекомендаций относительно агрегатора (набора агрегаторов) или элементов (набора элементов),

– извлечение элементов-кандидатов в порядке по убыванию рейтинга, рассчитанного рекомендательной системой.

Функционал библиотеки должен помещаться в оперативной памяти размером 16 гибибайт и отвечать на запросы в рекомендательном сервисе не более, чем за 200 миллисекунд.

Модели, распространяемые индексатором, должны быть размером менее 4 гибибайт. Процесс индексации не должен занимать более 20 минут.

Индексатор должен иметь возможность составления моделей одного из следующих типов:

– граф простых зависимостей (наличие-отсутствие),

– граф взвешенных зависимостей (при наличии зависимость может иметь выбранный индексатором вес),

– мульти-граф взвешенных зависимостей (между двумя сущностями может существовать более одной взвешенной связи).

После составления модели, граф может быть сериализован в файл или передан через интернет-подключение в формате CBOR.

Рекомендательный сервис в процессе подготовки должен иметь возможность десериализовать полученную от индексатора графовую модель.

Рекомендательный сервис осуществляет запросы к рекомендательному движку, используя один из следующих запросов:

– выбрать элементы, похожие на указанный элемент,

– выбрать элементы, похожие на указанный взвешенный набор элементов,

– выбрать элементы, похожие на зависимые от агрегатора,

– выбрать элементы, похожие на зависимые от взвешенного набора агрегаторов.

На вход индексатору поступают уникальные идентификаторы произвольного сериализуемого типа данных для элементов и агрегаторов.

Одна фиксированная графовая модель содержит не более десяти миллионов зависимостей между агрегаторами и элементами.

На вход рекомендательного движка поступают те же идентификаторы, которые были использованы при индексации модели.

Выходными данными являются идентификаторы рекомендуемых элементов, которые предоставляются в «ленивом» режиме для последующей фильтрации в порядке убывания рейтинга, присвоенного рекомендательным движком.

1.4 Концепция разрабатываемой рекомендательной системы

Одним из подходов, к которому обратились разработчики сервисов Pinterest [1] и Twitter [2], является построение рекомендательных систем на графовой модели. В основе этого исследования лежит концепция SALSA [8], использованная для составления рейтингов веб-страниц в системе PageRank. Модель системы представляется двудольным графом, в котором вершины одной доли ссылаются на вершины другой доли. При увеличении количества осуществленных переходов, вероятности посещения вершины стремятся к константе.

Одна из долей графа содержит рекомендуемые элементы в качестве своих вершин. Вершинами второй доли модели являются агрегаторы – элементы, группирующие рекомендуемые элементы на основе взаимодействий пользователей информационной системы. Приведем три примера информационных систем, различающие понятия пользователей, агрегаторов и рекомендуемых элементов. Разрабатываемая рекомендательная система интегрирована в каждый из описанных примеров.

В первом примере описывается графовая модель платформы, в которой пользователи просматривают размещенные на ней изображения и распределяют их в папки по некоторой тематике. В такой ситуации изображения являются рекомендуемыми элементами и персонализируются для пользователя. Папки являются агрегаторами и объединяют рекомендуемые элементы по содержимому, теме или другим признакам. Пользователи формируют свой профиль предпочтений, взаимодействуя с элементами и группируя их в своих целях. Таким образом, мы можем сформировать двудольный граф изображений и папок. Такой

платформой является мобильное приложение пиксельных раскрасок Sandbox Coloring App, в которой кроме пользовательских папок, изображения объединяются по тэгам и коллекциям, в которых они выпускаются (см.).

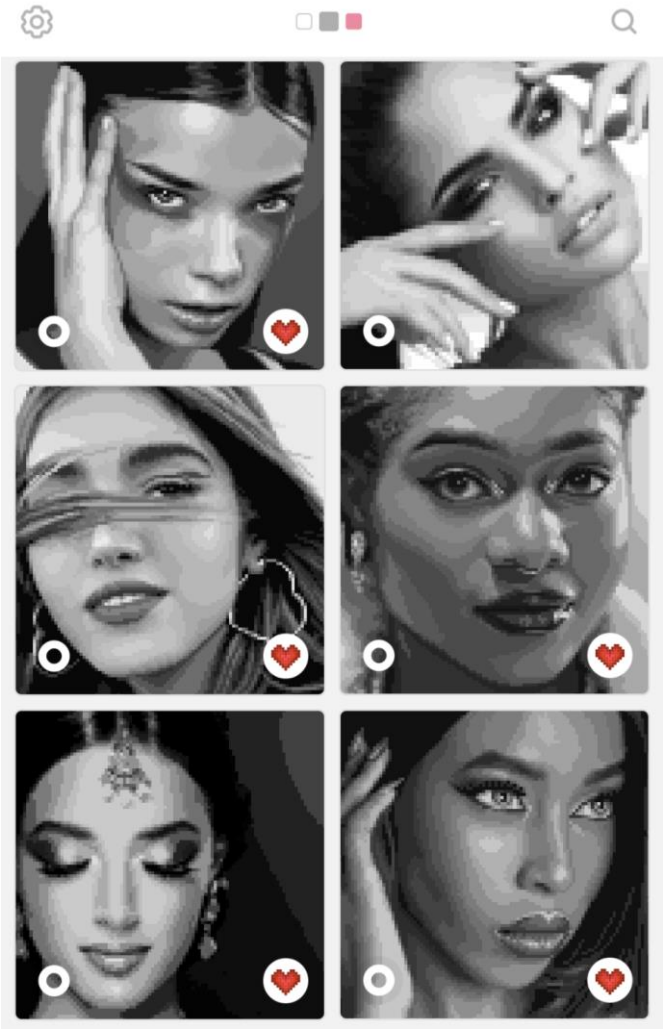


Рисунок 2 – Похожие пиксельные раскраски в приложении Sandbox Coloring App

Во втором примере представим платформу, в которой пользователи просматривают размещенные на ней изображения, делятся ими и отмечают наиболее понравившиеся элементы оценкой «нравится». В таком случае можно построить графовую модель, где агрегаторами являются сами пользователи, объединяющие нравящиеся им изображения в группы рекомендуемых элементов. В такой системе персонализация рекомендаций будет осуществляться по тому, насколько похожими будут группы элементов у разных пользователей. Примером такой платформы является Stich Pixel Art – мобильное приложение для просмотра пиксель-арта и поиска вдохновения (см. рисунок 3).

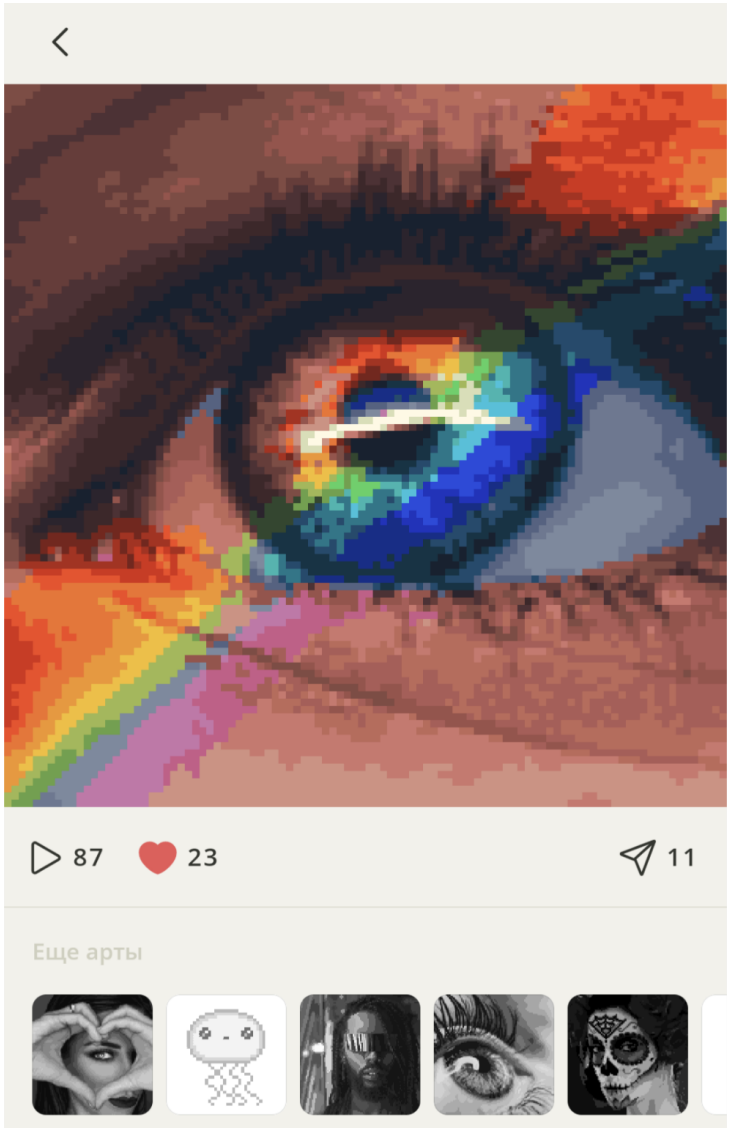


Рисунок 3 – Похожие пиксель-арты на платформе Stich Pixel Art

Заключительным примером является социальная сеть, в которой пользователи могут следить за публикациями пользователей и искать новых авторов для просмотра. Здесь требуется рекомендовать пользователей по тому, на каких пользователей уже осуществлена подписка. Таким образом, пользователь выступает и рекомендуемым элементом, и агрегатором, и субъектом, профиль которого определяется его текущими подписками [9]. В качестве альтернативы, можно также использовать публикации как агрегаторы пользователей, которые с ними взаимодействовали. Такая графовая модель характерна для социальной сети Vue – платформы микроблоггинга (см. рисунок 4).

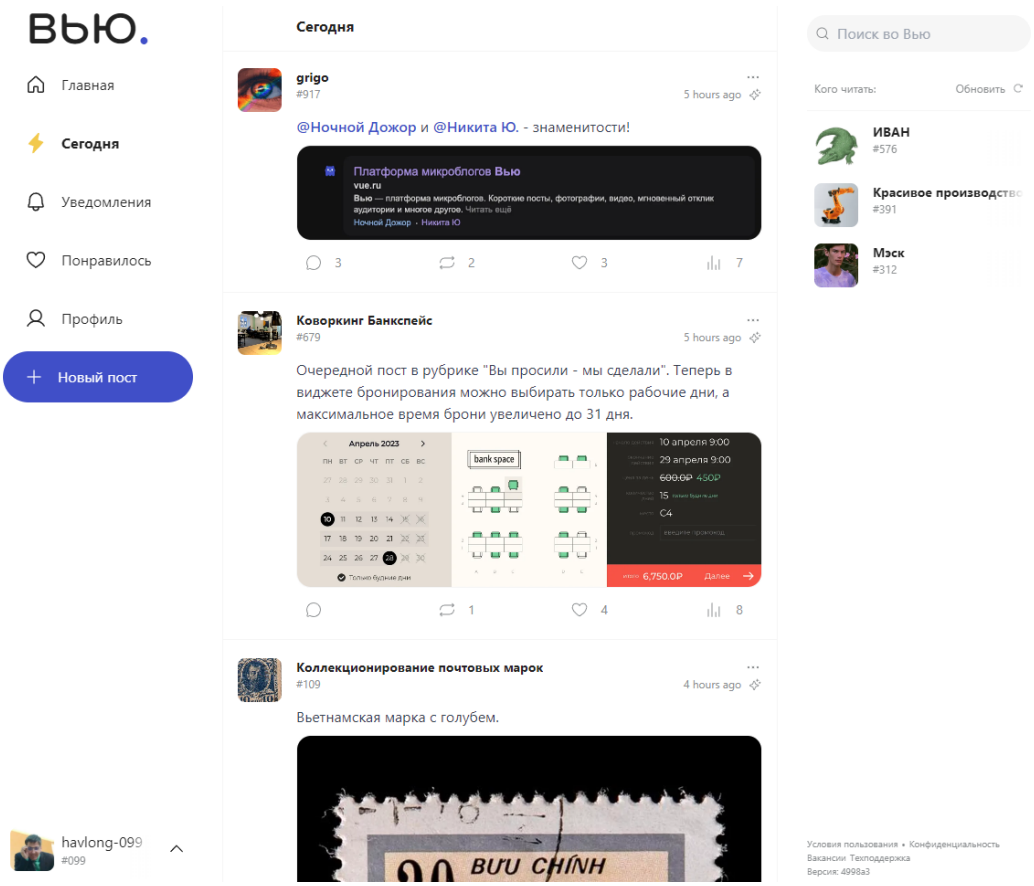


Рисунок 4 – Персонализированные рекомендации публикаций и пользователей на платформе микроблоггинга Vue

Ребра графовой модели определяются взаимодействиями пользователя с агрегаторами и рекомендуемыми элементами информационной системы. Назовем мульти-ребром множество ребер, состоящее более чем из одного элемента и соединяющее одну и ту же пару вершин. Граф рекомендательной системы является неориентированным, однако может быть взвешенным и иметь мульти-ребра.

Определим три вида моделей, которые применяются в рекомендательном движке:

– простая графовая модель. Не содержит взвешенных ребер и, предполагается, что она не содержит мульти-ребер, так как это повышает связность вершин по сравнению с остальными взаимодействиями,

– взвешенная графовая модель. Представлена взвешенным графом, в котором вес ребра эвристически отображает степень релевантности элемента по отношению к агрегатору. Граф при этом не содержит мульти-ребер,

– мульти-графовая модель. Представлена взвешенным графом, в котором присутствуют мульти-ребра, каждое из которых в целях оптимизации может быть сжато в одно взвешенное ребро, используя некоторую форму *бустинга* взаимодействий. Например, выбор максимального веса, выбор случайного из взаимодействий, суммирование весов мульти-ребра и другие.

Для первого из примеров, описанных выше, характерна простая графовая модель, поскольку взаимодействия гомогенны и уникальны для пары «агрегатор-рекомендация», так как в папку элемент может быть добавлен в единственном экземпляре и это – единственный тип взаимодействий в системе.

Для второго из примеров может быть применена взвешенная модель, так как имеются несколько типов взаимодействия между пользователем и элементом, которые можно оценить различным весом. Например, назначить больший вес связи пользователя с элементом, если он им делился, и меньший вес, если была поставлена отметка «нравится», избегая мульти-ребер при индексации графа.

Для третьего примера характерна модель мульти-графа, так как агрегаторами являются не только пользователи, но и публикации, с которыми пользователем осуществляются гетерогенные взаимодействия. В таком случае, можно назначить различные веса для связей «подписка на пользователя», «отметка нравится», «наличие ответной публикации», «перепубликация», «создание публикации». Мульти-ребра между пользователем и публикацией при этом будут сжаты моделью в одно ребро, используя один из видов *бустинга*, который может быть подобран экспериментально.

Используя эту концепцию, определим понятие случайного обхода модели. В качестве стартовой вершины выбирается один из элементов для рекомендации. На каждой итерации будем осуществлять два перехода из вершины в смежную вершину другой доли с вероятностью, линейно зависящей от веса ребра. Рейтинг посещенных вершин из доли рекомендуемых элементов при этом увеличивается на единицу. После каждой итерации с вероятностью α происходит возвращение в стартовую вершину. Этот процесс легко контролировать, так как время выполнения в основном зависит от количества итераций. После этого требуется

отсортировать элементы по рейтингу и выбрать требуемое количество кандидатов, имеющих наибольшие значения посещений в результате эмуляции. Итоговая асимптотика $O(T + M\ logM)$, где T – количество итераций случайного обхода, а M – количество рекомендуемых элементов модели.

В случае невзвешенного графа выбор ребра происходит вычислением случайного целого числа в полуотрезке $[0, r)$, где r – степень вершины, из которой осуществляется переход. Случайная величина определяет индекс смежного ребра, выбранного для осуществления перехода.

В случае взвешенного графа выбор ребра для перехода осуществляется следующим образом. Рассчитаем для вершины интегральные суммы весов смежных с ней ребер. Вычислим случайное значение в полуотрезке $[0, S)$, где S – общая сумма весов смежных ребер. Теперь найдем наибольшее значение интегральной суммы, не превышающее случайной величины. Ребро, соответствующее выбранной интегральной сумме, используется для перехода.

Графовая модель хранится и сериализуется в виде списка ребер, а также объектных идентификаторов вершин графа. Размер модели зависит от количества ребер в двудольном графе. В качестве ребра можно использовать взаимодействия пользователя с элементами, такие как просмотры, лайки и ответы, что приводит к гибкости и возможности использования системы в различных предметных областях. Требования к памяти описываются асимптотикой $O(E + N + M)$, где E – количество ребер, N – количество пользователей, M – количество элементов.

2 Разработка графовой рекомендательной системы

2.1 Графовые модели рекомендательной системы

Рекомендательная система состоит из двух основных компонентов: графовая модель, интерпретирующая контент в сервисе, и вариации обходчиков, которые используют граф для формирования рекомендаций. Подобная модель в рекомендательной системе Pixie представляет собой двудольный граф (см. рисунок 5), на котором вершинами одной доли (pins) является контент (рекомендуемые элементы), а вершинами другой доли (boards) являются коллекции этих элементов (агрегаторы).

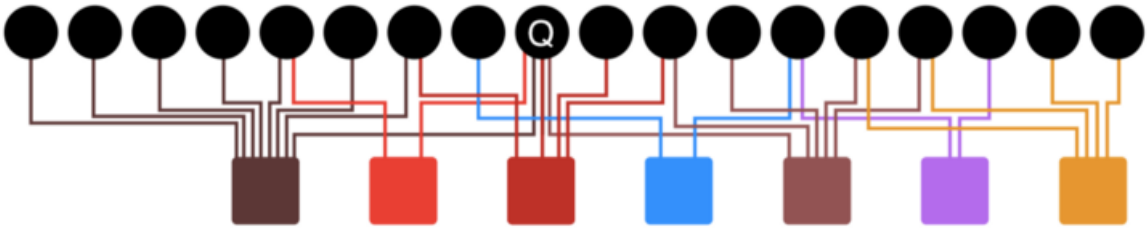


Рисунок 5 – Двудольный граф содержимого сервиса на базе Pinterest Pixie

В случае Pinterest, пользователи объединяли закладки с интересующей их информацией на досках. Преимуществом же разработки является ее универсальность и возможность гибкой адаптации модели к разным приложениям.

Согласно вышеописанной концепции рекомендательной системы (см. подраздел 1.4), разрабатываемая библиотека поддерживает три вида моделей:

– простая графовая модель (см. рисунок 6),

– взвешенная графовая модель (см. рисунок 7),

– мульти-графовая модель (см. рисунок 8).

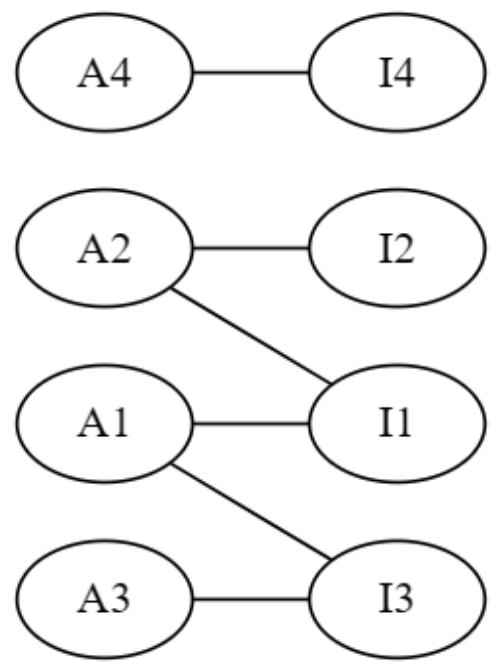


Рисунок 6 – Пример простой графовой модели

Описание простой графовой модели состоит из:

– множества рекомендуемых элементов (и их ID в системе),

– множества агрегаторов (и их ID в системе),

– множества уникальных пар «элемент-агрегатор».

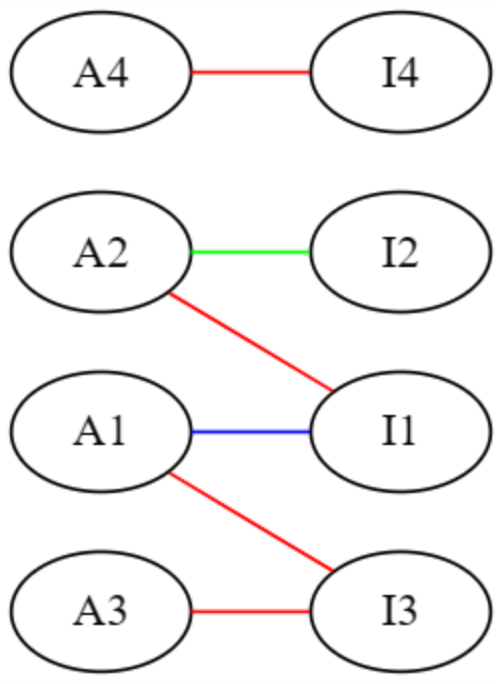


Рисунок 7 – Пример взвешенной графовой модели

В случае взвешенной графовой модели изменяется вид ребер, которые теперь могут иметь вес положительного значения.

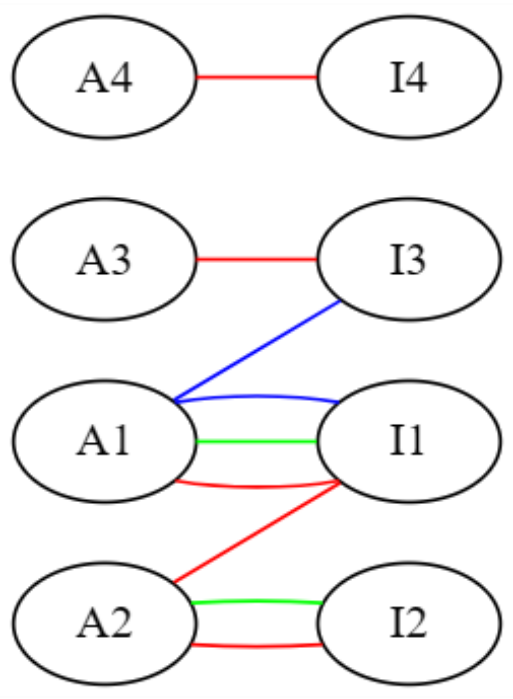


Рисунок 8 – Пример мульти-графовой модели

В случае мульти-графа, после индексации рекомендательный движок дает разработчику выбор: оптимизировать граф, сжав мульти-ребра в единое взвешенное ребро, или оставить граф без изменений, экспортировав его с неуникальными парами «элемент-агрегатор». Полная диаграмма классов для графовой модели приведена в приложении А.2.

2.2 Организация структуры хранения данных графовых моделей

Ребра графовой модели могут быть представлены двумя типами: взвешенные и невзвешенные (простые). Введем интерфейс ребра добавим реализацию для каждого из типов.

Интерфейс PixieEdge состоит из трех неизменяемых свойств:

– l – вершина из первой доли двудольного графа, представленная индексом из списка вершин первой доли,

– r – вершина из второй доли двудольного графа, представленная индексом из списка вершин второй доли,

– w – вещественный вес ребра, соединяющего вершины.

В случае простого ребра, вес вершины представлен константой DEFAULT_WEIGHT, равной единице. (см. рисунок 9)

Такая организация хранения ребер позволяет организовать перекрестную совместимость моделей и алгоритмов. Иными словами, интерфейс позволяет использовать взвешенные графы в простых обходах без использования весов, а также удобно конвертировать простые графы во взвешенные для добавления новых ребер и применения обходов, учитывающих веса.

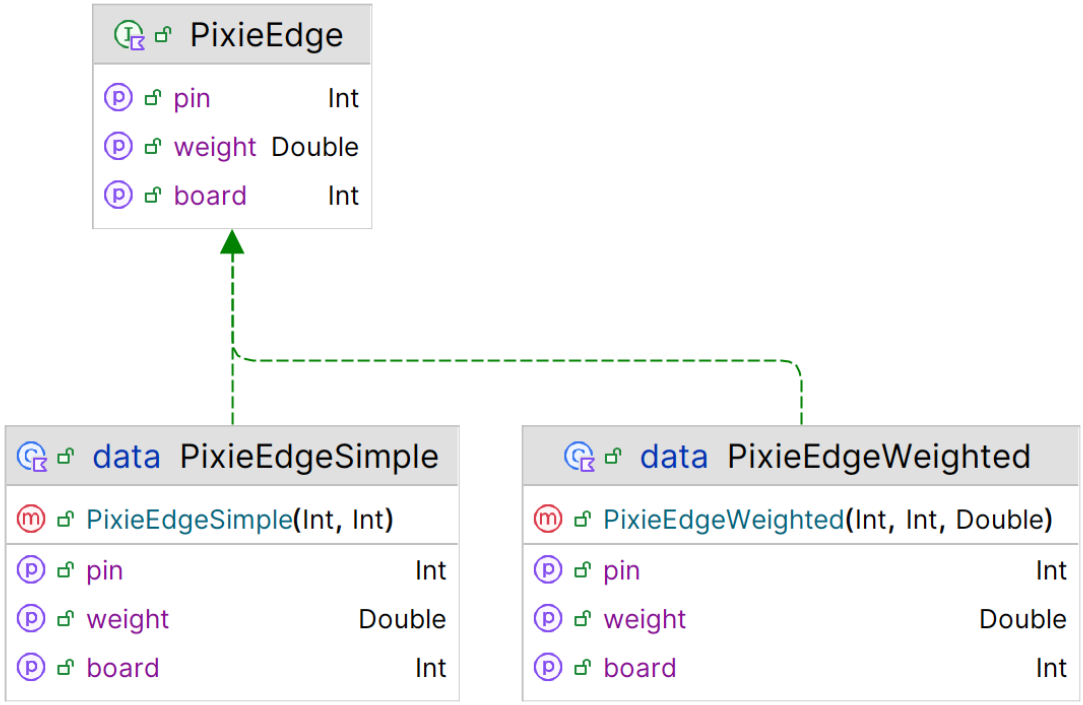


Рисунок 9 – Диаграмма классов, описывающих ребро графа

Так как в процессе обхода графа информация о вершинах и соответствующих идентификаторах не используется, удобно выделить эти данные в отдельный компонент. Назовем такой компонент дескриптором (см. рисунок 10), и перенесем на него ответственность за двустороннее преобразование идентификаторов объектов и индексов вершин графа. Типы идентификаторов в различных приложениях могут отличаться. Так, в качестве идентификатора может выступать число, хэш-код, UUID, ObjectID СУБД MongoDB, сложный ключ из нескольких элементов и даже просто строка. Тогда, нашей системе потребуется использовать два типа данных: P – объектный тип

данных идентификатора рекомендуемых элементов (PinType) и B – объектный тип данных идентификатора доски (BoardType).

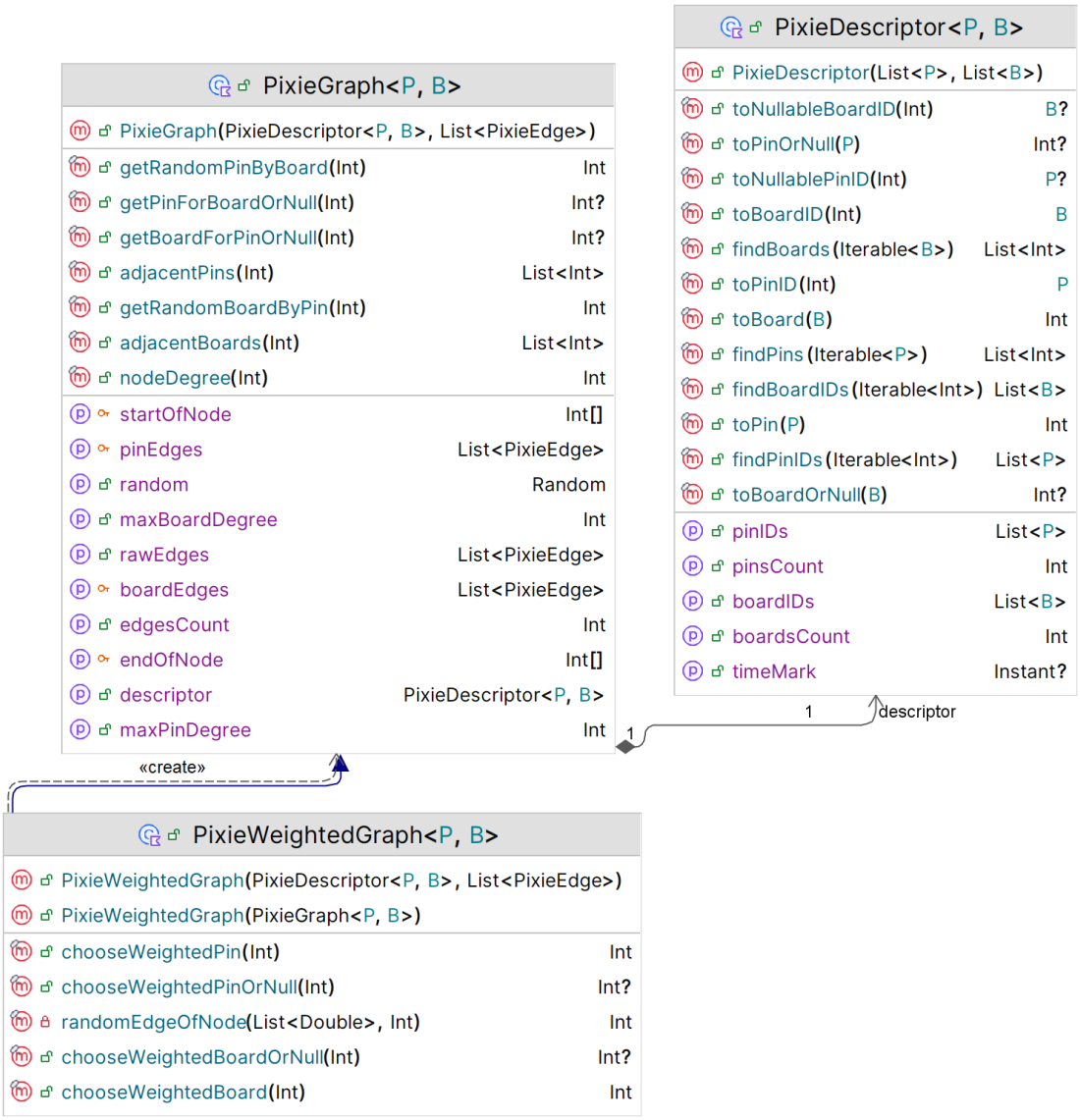


Рисунок 10 – Диаграмма классов графовой модели рекомендательной системы

Кроме данных о вершинах графа, дескриптор также может сохранять временную отметку создания графовой модели для последующего принятия сервером решения об использовании более свежей модели.

Структура графа определяется двумя классами для случаев обработки запросов, в которых граф должен быть невзвешенным или взвешенным. Весь функционал невзвешенного графа свойственен и взвешенному, поэтому имеет смысл наследовать класс PixieWeightedGraph от класса PixieGraph.

Дескриптор и невзвешенный граф находятся в отношении композиции как содержимое и контейнер соответственно (см. рисунок 10). Кроме дескриптора, используемого для подготовки запросов к графу, класс содержит сырой список ребер, который при инициализации объекта распределяется по спискам boardEdges и pinEdges, отсортированным по индексам агрегаторов и элементов соответственно. Это позволяет за константную асимптотику извлекать список смежных ребер для вершины, используя индексы startOfNode и endOfNode. По сути, степень вершины вычисляется как endOfNode[v] – startOfNode[v]. Зная тип вершины, можно обратиться к списку ребер этого типа, и тогда ребра с индекса startOfNode включительно до endOfNode будут относиться к этой вершине. Кроме того, после инициализации графовая модель содержит в себе конфигурируемый объект генератора псевдослучайных чисел из-за того, что именно этот компонент выбирает случайные смежные вершины для заданных.

В случае взвешенного графа, расширяющего класс PixieGraph (см. рисунок 10), требуется обрабатывать запросы выбора смежных ребер с учетом их весов. Для этого согласно вышеописанной концепции, применяются интегральные суммы, которые лениво вычисляются при инициализации взвешенного класса.

2.3 Процесс индексирования модели рекомендательной системы

При индексировании моделей информационная система использует предоставленные библиотекой редактор. Графовая модель сама по себе является неизменяемой, и преобразования с изменениями осуществляют классы SimplePixieEditor, WeightedPixieEditor и BoostedPixieEditor, которые инициализируются кроме пустого конструктора имеют конструктор, получающий в качестве аргумента объект PixieGraph (см. рисунок 11). При этом, SimplePixieEditor отвечает за инициализацию и экспорт простой графовой модели, WeightedPixieEditor управляет взвешенной графовой моделью, а BoostedPixieEditor используется в случае мульти-графов, когда между вершинами присутствуют мульти-ребра.

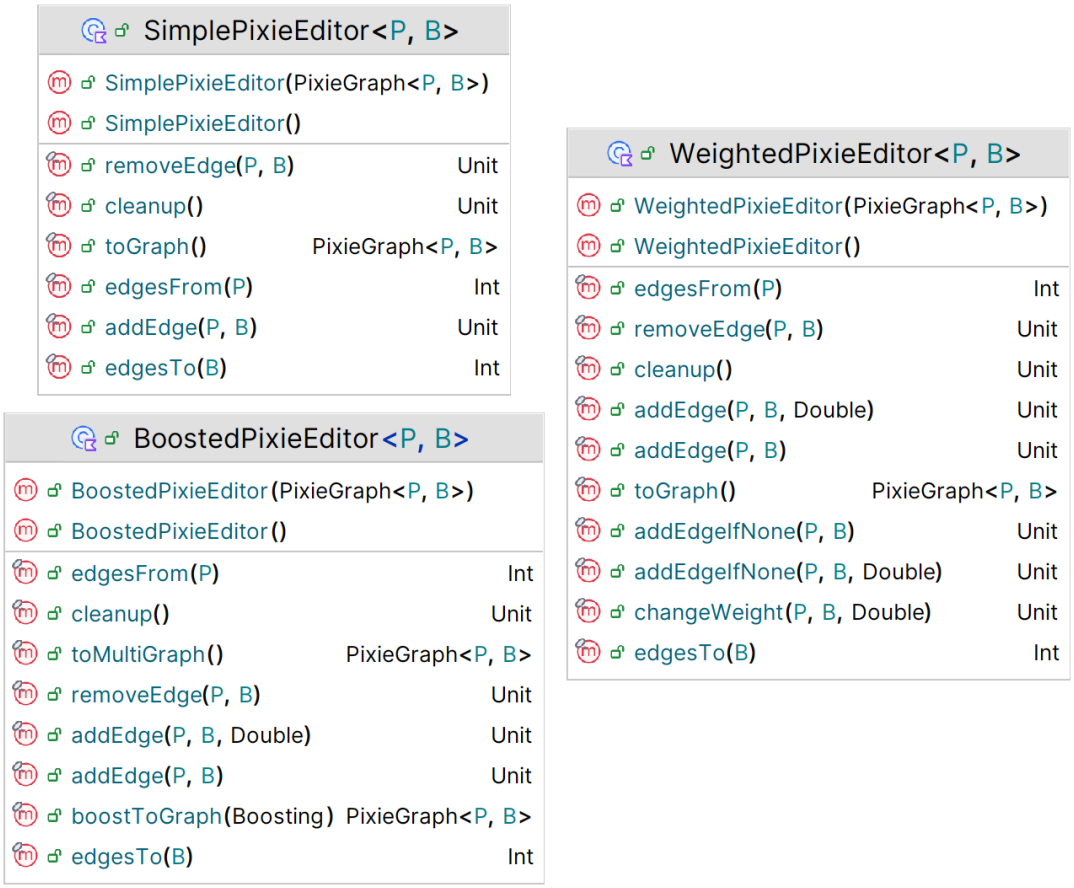


Рисунок 11 – Диаграмма классов редакторов графовых моделей

SimplePixieEditor хранит уникальные множества рекомендуемых элементов, агрегаторов и ребер между ними. Для управления ребрами используются функции addEdge и removeEdge. Для получения информации о количестве ребер из элемента к агрегаторам присутствует функция edgesFrom, а обратная функция – edgesTo сообщает о количестве ребер из элементов к агрегатору. Метод toGraph осуществляет экспорт сформированной графовой модели из простых ребер PixieEdgeSimple.

WeightedPixieEditor используется для работы со взвешенными графами и помимо связываемых вершин, получает в качестве аргумента вес добавляемого ребра. В случае присутствия ребра метод addEdge заменяет вес на переданный в параметре. Для того, чтобы не изменять вес в случае присутствия ребра, реализован метод addEdgeIfNone. Метод changeWeight, в свою очередь, используется для изменения веса ребра без добавления нового ребра в случае его отсутствия.

BoostedPixieEditor позволяет информационной системе генерировать и редактировать мульти-графовые модели. Из-за неудобства обработки множества весов ребер между вершинами, редактор не позволяет выбирать и удалять ребра по отдельности и метод removeEdge удаляет мульти-ребро целиком. Отличием является наличие двух методов экспорта результирующего графа со взвешенными ребрами: toMultiGraph оставляет граф без изменений, сохраняя множественные ребра, а boostToGraph позволяет сжать мульти-ребро в единственное с помощью произвольной функции бустинга. В качестве стандартных библиотека предлагает следующие:

– сохранить ребро максимального веса,

– сохранить ребро минимального веса,

– создать ребро с суммарным весом,

– использовать первое добавленное ребро между вершинами,

– использовать последнее добавленное ребро между вершинами,

– выбрать случайное ребро между вершинами.

С использованием этих редакторов, использующая рекомендательный движок система может подготовить конвейер, в котором периодически запускается процесс индексации, модель сериализуется и отправляется сервисам для обработки запросов к рекомендательной системе.

Типичным сценарием в условиях пространственных ограничений на сервере является индексация модели случайными взаимодействиями из базы данных. Например, MongoDB имеет агрегатор sample, позволяющий выбрать некоторое количество элементов в случайном порядке. Для увеличения доли свежих рекомендаций также рекомендуется применять шкалирование весов взаимодействий по новизне, увеличивая присутствие недавней активности в рекомендательной системе.

2.4 Подготовка моделей системы к горизонтальному масштабированию

Использование языка программирования Kotlin для реализации рекомендательного движка способствует динамической сериализации модели в

экономный формат. Для этих целей была выбрана библиотека «kotlinx.serialization», которая позволяет использовать код для сериализации, генерируемый с помощью рефлексии на этапе компиляции. Из-за того, что основным критерием качества сериализации является объем итоговых данных, требуется использовать бинарный протокол сериализации. В качестве альтернативы рассматривался Google ProtoBuf, при работе с которым возникли проблемы с генерацией кода.

Разделим форматы данных для хранения и передачи на три группы: человекочитаемые форматы (такие как JSON или CSV), схемонезависимые бинарные форматы (такие как CBOR и MsgPack) и схемозависимые бинарные форматы (такие как ProtoBuf). Недостаток схемозависимых форматов состоит в том, что они зависят от ряда факторов, таких как платформа исполнения, реализация сериализации, наличие схемы данных на всех клиентах. При этом, применение этих форматов в проекте позволяет получить преимущество в виде размера сериализованных данных и скорости выполнения алгоритмов над ними. Рассмотрим поддерживаемые kotlinx.serialization форматы сериализации [10]:

– JSON - человекочитаемый формат, наиболее популярный при взаимодействии с веб-сервисами. Библиотека предоставляет стабильную версию сериализации в этом формате,

– HOCON - более гибкий аналог JSON. Библиотека предоставляет экспериментальную реализацию сериализации в этом формате,

– CBOR - бинарный схемонезависимый формат, являющийся расширением формата MsgPack. Библиотека предоставляет экспериментальную реализацию сериализации в этом формате,

– MsgPack - бинарный схемонезависимый формат. Лозунг - как JSON, но быстро и компактно. Присутствует реализация, построенная сообществом,

– Apache Avro - позволяет передавать данные в бинарном и JSON форматах. Присутствует реализация, построенная сообществом,

– Bson - формат сериализации, используемый в MongoDB. Присутствует реализация, построенная сообществом.

В качестве дополнительного критерия выбора формата введем поддержку иерархических зависимостей, так как ребра в графе могут быть представлены одним из двух классов, что дает выигрыш в итоговом размере модели при использовании простых ребер. Среди предоставленных библиотекой форматов, наиболее подходящим является схемонезависимый CBOR, который является бинарным.

При использовании формата для сериализацией была обнаружена и решена проблема обработки графовых моделей, сериализованный объем которых имел размер более одного гибибайта. Проблема возникла из-за деталей реализации JVM, где байтовые массивы нельзя инициализировать размером более одного из-за максимального количества элементов у произвольного массива. Проблема была решена модификацией исходного кода библиотеки и добавлением сериализации с помощью потоков данных. Так, данные графовой модели можно сереализовать сразу в файл на диске или отправлять через сетевое соединение.

2.5 Предлагаемый алгоритм анализа элементов

Вторым основным компонентом в системе являются запросы пользователей, которые эмулируются стохастическими обходами графа с учетом весов. Существует два типа запросов по модели рекомендательной системы Pinterest Pixie [1]: простые и мульти-вершинные.

Представим процесс поиска пользователем интересующего его контента в виде марковского процесса. Тогда, вершины двудольного графа являются состояниями, а его ребра – переходами. Если представить вершину (вершины) запроса в качестве начального состояния, то при увеличении количества переходов, вероятности состояний процесса будут сходиться, формируя рейтинги рекомендуемого контента для вершин доли кандидатов. Из-за огромного размера графа и соответствующей ему матрицы, переходы тяжело рассчитать, и более осуществимым подходом является прямая эмуляция вероятностного обхода графа.

Для простых запросов осуществляется простой стохастический обход графа. При этом при посещении вершин доли рекомендуемых элементов (кандидатов),

их рейтинг увеличивается на единицу, а текущая вершина с вероятностью, описанной параметром α, может быть сброшена в начальную [8]. По истечению количества итераций, выделенного на выполнение обхода, вектор рейтингов фиксируется. Рекомендуемые вершины выбираются по убыванию рейтинга и продвигаются по конвейеру системы в ранкер или напрямую в миксер.

Для мульти-вершинных обходов будем запускать обходы на каждой из вершин запроса, распределив общее количество итераций между ними. Будем регулировать долю операций на вершину с помощью подхода Pinterest Pixie. Для каждой из вершин запроса рассчитывается величина s (см. рисунок 12), которая совмещает степень вершины, но смягчает падение критерия при низких значениях с помощью логарифма.

$$s_q = |E(q)| \cdot (C - \log |E(q)|)$$

Рисунок 12 – Формула степенного критерия для вершины в мульти-запросах Pinterest Pixie

Затем доля от общего степенного критерия умножается на вес вершины в мульти-запросе и этот коэффициент умножается на общее количество итераций (см. рисунок 13).

$$N_q = w_q N \frac{s_q}{\sum_{r \in Q} s_r}$$

Рисунок 13 – Формула количества итераций простого обхода на вершину в мульти-запросе Pinterest Pixie

После завершения всех дочерних простых обходов, результирующий рейтинг вершины вычисляется с помощью бустинга (см. рисунок 14). При таком объединении результатов, посещение множеством обходов, увеличивает рейтинг вершины, отмечая ее релевантность.

$$V[p] = \left(\sum_{q \in Q} \sqrt{V_q[p]} \right)^2$$

Рисунок 14 – Формула бустинга рейтингов для вершины в мульти-запросе Pinterest Pixie

При этом сумма итераций, сделанных на подзапросах, является фиксированной и управляемой, что ограничивает время последовательного их выполнения.

2.6 Оптимизация скорости исполнения запросов

Перед параллелизацией запросов системы стоит несколько задач:

– синхронизация фрагментов памяти, в которых накапливаются рейтинги (или использование потокобезопасных структур),

– ограничение памяти с повторным использованием,

– массовый запуск подзапросов мульти-вершинного запроса.

Задачи были решены с использованием экосистемы сопрограмм языка Котлин [11]. Для решения первых двух задач прекрасно подходит аналог асинхронной очереди – каналы. На старте объекта-менеджера рекомендательной системы в соответствии с параметрами инициализируется диспетчер сопрограмм поверх пула потоков, а также создаются обходчики (Walkers) – структуры данных, аккумулирующие рейтинги, помещающиеся в каналы. Запросы к рекомендательной системе блокируются на очереди до получения обходчика, после чего на нём проводятся вычисления, и обходчик возвращается в свой канал объекта-менеджера.

Теперь, имея диспетчер и объекты для выполнения обходов, параллелизация становится простой задачей. Конвертируем функции запросов в корутины, которые будут выполняться на диспетчере сопрограмм объекта-менеджера [12]. При простом запросе, займем обходчик, выполним запрос и вернем обходчик в соответствующий канал, а результат в качестве отложенного результата. При сложном запросе, займем обходчик, объединяющий результаты подзапросов, и массово запустим корутины простых подзапросов, последовательно объединяя их

в момент выполнения и сразу же возвращая их в канал простых обходчиков [12]. Процесс выполнения мульти-вершинного запроса изображен на рисунке 15.

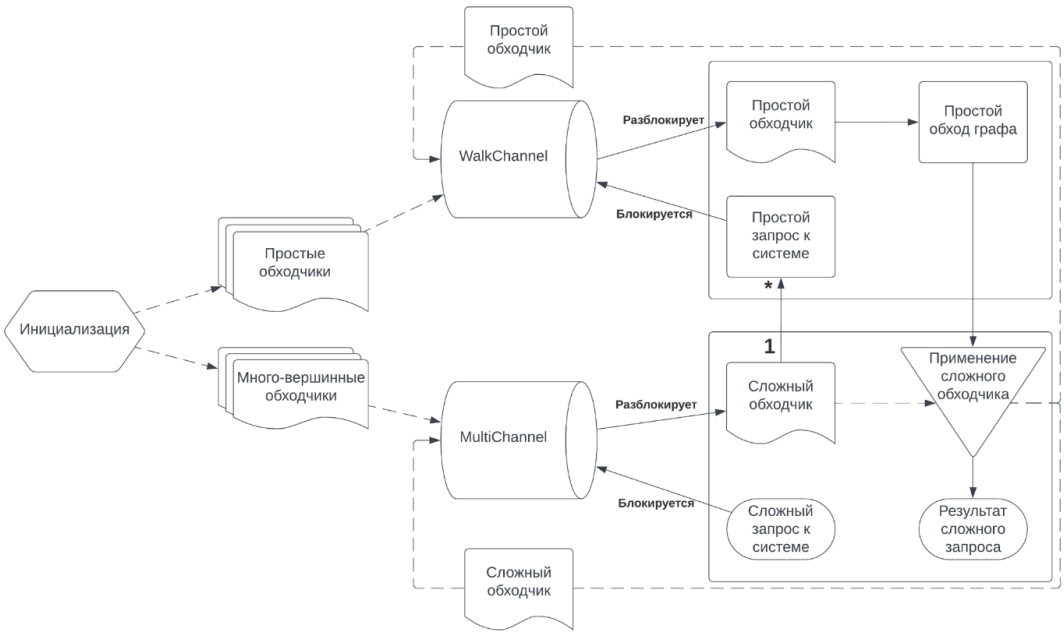


Рисунок 15 – Схема выполнения сложного запроса в рекомендательной системе

В результате применения стратегии Разделяй-и-Властвуй, преобразование простых запросов в корутины существенно упростило распараллеливание сложных запросов, так как объекты с данными простых запросов распределены между сопрограммами взаимоисключающим образом.

3 Оценка результатов исследования

Тестирование рекомендательных систем – одна из самых главных проблем из-за неоднозначности понятия «качество». Существует два основных подхода к тестированию:

– оффлайн-тестирование модели на исторических данных с помощью ретро-тестов,

– тестирование разработанной модели с помощью A/B тестирования.

3.1 Тестирование совместимости

Одним из основных требований, выдвинутых перед разработкой, является совместимость с языком Java. На рисунке 16приведен скриншот одного из тестов совместимости рекомендательной системы с языком Java.

```
@Test  havlong *
void testQueryFromJava() {
    Deferred<PixieVisitWalker> deferredWalker = queryManager.walkSimple(
            PIXIE_GRAPH,
            QUERY_NODE,
            MAX_STEPS,
            RESET_PROBABILITY
    );

    List<String> recommended = new ArrayList<>();

    try (PixieVisitWalker walker = awaitFuture(deferredWalker)) {
        walker.collectPins(PIXIE_GRAPH).forEachRemaining(recommended::add);
        System.out.println(recommended.size());
    } catch (IllegalArgumentException exception) {
        exception.printStackTrace();
    }
}
```

Рисунок 16 – Скриншот теста вызова рекомендательного движка из языка Java

В приведенном сценарии осуществляется обход графовой модели из статического поля PIXIE_GRAPH простым асинхронным обходчиком на сконфигурированном объекте queryManager. Поскольку синхронное ожидание результата является частью языка Kotlin, в библиотеке присутствует функция awaitFuture, ожидающая завершения выполнения запроса и извлекающая

обходчик PixieVisitWalker, в котором находятся результаты доступные для ленивого поглощения из итератора.

3.2 Тестирование корректности

Модульное тестирование корректной работы включает в себя:

– стохастическое тестирование генерации графовых моделей,

– стохастическое тестирование запросов к рекомендательной системе,

– асинхронное тестирование, проверяющее корректность суммы рейтингов в каждом из запросов,

– тестирование сериализации и десериализации модели с проверкой содержимого,

– тестирование совместимости рекомендательной системы с объектными идентификаторами.

На рисунке 17приведен пример стохастической генерации графовых моделей. Этот граф используется при конструировании простого редактора, взвешенного редактора, и мульти-графового редактора в сценарии на рисунке 18.

Граф использует объектные идентификаторы MongoDB, которые также применяются при тестировании совместимости рекомендательной системы со сторонними объектными идентификаторами

В сценарии генерируется граф сравнительно небольшого размера с целью сравнения работы оригинальной библиотеки kotlinx.serialization.cbor и ее модификации, предоставляющей работу с потоками для сохранения в файл и передачи через сетевое соединение.

Генерируемый граф является взвешенным мульти-графом и имеет веса случайной положительной величины. При экспорте мульти-графа применяется бустинг, сохраняющий среди добавленных ребер лишь ребро с максимальным весом.

```kotlin
@BeforeAll
fun setup() {
    val editor = BoostedPixieEditor<ObjectId, ObjectId>()

    val random = ThreadLocalRandom.current().asKotlinRandom()

    val pinAmount = 200_000
    val boardAmount = 100_000
    val edgesAmount = 500_000

    val pins = buildSet { repeat(pinAmount) { add(ObjectId.get()) } }.toList()
    val boards = buildSet { repeat(boardAmount) { add(ObjectId.get()) } }.toList()

    val qBoards = mutableSetOf<ObjectId>()
    val qPins = mutableSetOf<ObjectId>()

    repeat(edgesAmount) {
        val l: ObjectId = pins.random(random)
        val r: ObjectId = boards.random(random)

        editor.addEdge(l, r,  weight: random.nextDouble( until: 5.0) + 0.1)

        qPins.add(l)
        qBoards.add(r)
    }

    queryablePins = qPins
    queryableBoards = qBoards

    pixieGraph = PixieWeightedGraph(editor.boostToGraph(MAX))
```

Рисунок 17 – Скриншот стохастической генерации графа

```kotlin
@Test  havlong
fun boostedGraphConstructionTest() {
    assertDoesNotThrow(::setup)

    assertEquals(pixieGraph.descriptor.pinsCount, SimplePixieEditor(pixieGraph).pinsCount)
    assertEquals(pixieGraph.descriptor.boardsCount, SimplePixieEditor(pixieGraph).boardsCount)
    assertEquals(pixieGraph.edgesCount, SimplePixieEditor(pixieGraph).edgesCount)

    assertEquals(pixieGraph.descriptor.pinsCount, WeightedPixieEditor(pixieGraph).pinsCount)
    assertEquals(pixieGraph.descriptor.boardsCount, WeightedPixieEditor(pixieGraph).boardsCount)
    assertEquals(pixieGraph.edgesCount, WeightedPixieEditor(pixieGraph).edgesCount)

    assertEquals(pixieGraph.descriptor.pinsCount, BoostedPixieEditor(pixieGraph).pinsCount)
    assertEquals(pixieGraph.descriptor.boardsCount, BoostedPixieEditor(pixieGraph).boardsCount)
    assertEquals(pixieGraph.edgesCount, BoostedPixieEditor(pixieGraph).edgesCount)

    val edges: List<PixieEdge> = pixieGraph.rawEdges
    assertEquals(edges.size, edges.map { it.pin to it.board }.distinct().size)
}
```

Рисунок 18 – Скриншот тестирования редакторов графовых моделей

Для импортированных в редакторы графов проверяется количество вершин в каждой из долей, а также количество ребер между уникальными парами вершин.

Для тестирования обходов применяются различные сценарии, включая запросы от несуществующих в графе элементов (см. рисунок 19).

```
@Test  havlong
@Tag("walk")
fun queryNotExistentPinTest() {
    val queryPin = ObjectId.get()
    assertFalse { queryPin in queryablePins }

    lateinit var resultList: Map<ObjectId, Double>

    assertDoesNotThrow {
        resultList = buildMap {
            awaitFuture(pixieManager.walkSimple(objectIdGraph, queryPin, querySteps,  alpha: 0.4))
                .let { walker ->
                    walker
                        .collectScored(objectIdGraph)
                        .forEachRemaining { (id, score) -> if (score > 0) this[id] = score }
                        .also { pixieStorage.unlockVisitWalker(walker) }
                }
        }
    }

    assertTrue { resultList.isEmpty() }
}
```

Рисунок 19 – Скриншот тестирования простого обхода

Проверяется отсутствие исключений при вызове библиотеки и пустота результатов для рекомендации клиенту. При этом, дескриптор графа работает с идентификаторами базы данных MongoDB.

Для тестирования корректной работы асинхронных запросов, реализованы тесты, которые вызывают рекомендательный движок из нескольких потоков. Для этого используются механизмы асинхронного программирования Java, а сами запросы ожидаются синхронно на нескольких потоках. (см. рисунок 20)

Также был протестирован процесс сериализации графовой модели с проверкой отсутствия новых связей и сохранности количества зависимостей после процесса десериализации. (см. рисунок 21)

Аналогичные тесты были разработаны для сериализации с использованием формата JSON и неотредактированной библиотеки для работы с форматом CBOR с целью сравнения корректности данных, скорости выполнения и результирующего размера графовой модели. Сравнение метрик скорости и размера присутствует в подразделе 3.3.

```
    fun multiThreadQueryTest() {
        val threadPool = Executors.newFixedThreadPool( nThreads: threadsToStart / 2)

        val threadTask: () -> Int = {
            val graph = pixieGraph
            val qPin = pins.random()

            val walker = runBlocking {
                pixieManager.walkSimple(
                    graph,
                    qPin,
                    querySteps,
                    alpha: 0.5
                ).await()
            }

            walker.nodeIterator()
                .asSequence()
                .sumOf { (_, score) -> score.toInt() }
                .also { pixieStorage.unlockVisitWalker(walker) }
        }

        val ratingSums = assertDoesNotThrow {
            List(threadsToStart) { CompletableFuture.supplyAsync(threadTask, threadPool) }
                .map(CompletableFuture<Int>::get)
        }

        ratingSums.forEach { ratingSum ->
            assertEquals(querySteps, ratingSum)
        }
    }
```

Рисунок 20 – Скриншот сценария отправки запросов из многопоточной асинхронной среды

```
@Test
fun customCborTest() {
    val tempFile = File.createTempFile( prefix: "graph_pixie_cbor",  suffix: ".bin")
    val serializer = PixieCborSerialization

    assertDoesNotThrow {
        serializer.serializeToFile(graph, tempFile)
        logger.info( msg: "Serialized graph is ${tempFile.length()} bytes long")
    }
    val deserialized: PixieWeightedGraph<String, String> =
        PixieWeightedGraph(serializer.deserializeFromFile(tempFile))

    assertEquals(graph!!.edgesCount, deserialized.edgesCount)
    assertThrows(NullPointerException::class.java) { deserialized.descriptor.toPin( pinID: "IF") }

    val pin: Int = deserialized.descriptor.toPin( pinID: "I_F")
    val randomBoardByPin = deserialized.getRandomBoardByPin(pin)
    assertNotNull(randomBoardByPin)
    val randomBoard: String = deserialized.descriptor.toBoardID(randomBoardByPin)

    assertTrue(listOf("C_B", "C_C", "C_E").contains(randomBoard))
}
```

Рисунок 21 – Скриншот теста сериализации модели

С использованием средств среды разработки IntelliJ Idea было вычислено покрытие библиотеки тестами, отображенное на рисунке 22.

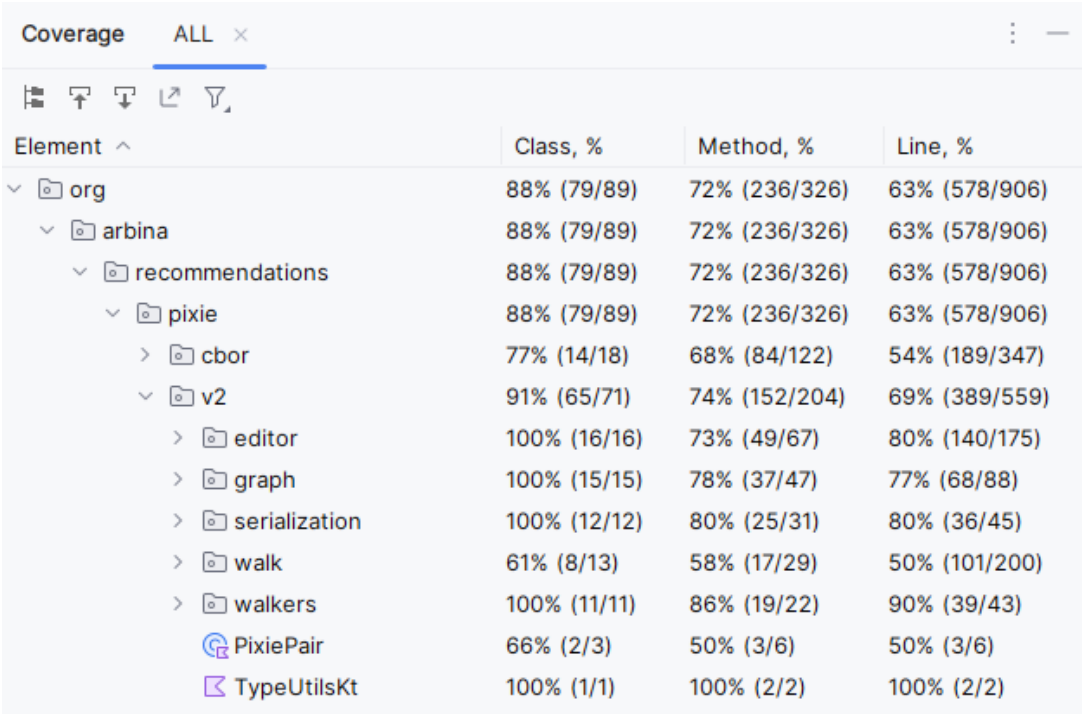


Рисунок 22 – Скриншот вычисленного покрытие компонентов библиотеки

В результате выполнения метрика покрытия тестами разработанного рекомендательного движка составила 69%. Разработанные модульные тесты отработали успешно. Результаты выполнения тестирования отображены на рисунке 23.

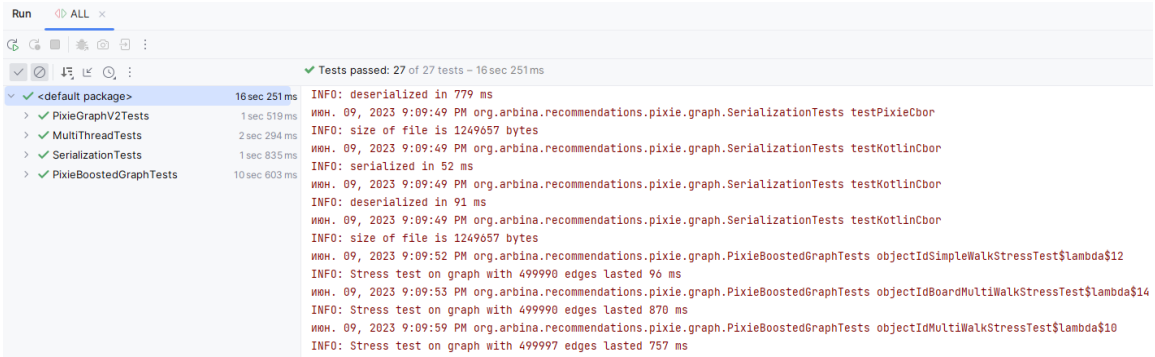


Рисунок 23 – Скриншот успешного выполнения тестов библиотеки

Тестирование отобразило корректность исполнения сценариев, проверяющих назначенные функциональные и технические требования к разработанному программному обеспечению.

3.3 Тестирование эффективности

В таблице представлены результаты стресс-тестирования системы, получающей восемь запросов в четырех потоках для случайного графа из 10^6 ребер, $2*10^5$ кандидатов и 10^5 агрегаторов. В менеджере были инициализированы 4 мульти-вершинных и 50 простых обходчиков. Лимит на выполнение запросов согласно требованиям составляет 1800 мс.

Таблица 3 – Результаты тестирования параллелизации

Количество потоков в пуле менеджера	1 поток	2 потока	4 потока	6 потоков
Корректность суммы итераций (рейтингов)	да	да	да	да
Время простых запросов	184 мс	141 мс	97 мс	113 мс
Время простых запросов без извлечения рейтингов	162 мс	92 мс	71 мс	75 мс
Время многовершинных запросов из 100 вершин	3191 мс	1774 мс	1332 мс	1418 мс

Кроме того, в таблице был отмечен факт прохождения теста на потокобезопасность. Это проверка отсутствия взаимного вмешательства обходчиков в процессе мульти-вершинного и простого обходов, что констатируется различием между суммой рейтингов вершин и выделенным на запрос количеством итераций. В результате тестирования лучшие результаты были достигнуты при оптимальных для машины четырех потоках.

При тестировании различных форматов сериализации использовался стохастически генерируемый взвешенный граф. Рассматривались сценарии для сериализации в форматы JSON и CBOR, где JSON обрабатывается библиотекой kotlinx.serialization.json, а формат CBOR обслуживается модификацией, совместимой с потоками для записи графовой модели в файл. Результаты приведены в таблице .

Под пинами понимаются вершины в доле с кандидатами рекомендуемых элементов. Под досками понимаются вершины в доле с агрегаторами. Ребра между пинами и досками уникальны.

Таблица 4 – Результаты тестирования форматов сериализации при различных размерах графа

№ теста	Количество пинов	Количество досок	Количество ребер	Размер	Время сериализации	Время десериализации
1 JSON	20’000	10’000	50’000	3 030,6 КБ	545 мс	370 мс
1 CBOR				1 626,6 КБ	114 мс	979 мс
2 JSON	20’000	10’000	1’000’000	33,318 МБ	7740 мс	3041 мс
2 CBOR				17,011 МБ	421 мс	9856 мс
3 JSON	40’000	70’000	20’000	2 283,2 КБ	324 мс	260 мс
3 CBOR				1 326,5 КБ	103 мс	892 мс

В результате сравнения метрик, CBOR имеет результаты в размере итоговой модели почти в два раза лучше, чем JSON. При сериализации работает быстрее, но проигрывает при их десериализации. Как упоминалось ранее, ведущей метрикой при распространении моделей был установлен размер сериализованной модели, что обосновывает выбор CBOR для хранения и передачи.

Ограничения по памяти, занимаемой моделями, согласно проведенному тестированию, удовлетворены. Механизмы системы работают без ошибок. Для дальнейшей оценки разработанного рекомендательного движка оффлайн-тестирования недостаточно и требуется проведение экспериментов над различными моделями.

3.4 Методология маркетингового исследования при внедрении

Применение ретро-тестирования к движку на графовых моделей проблематично, так как процесс оценки кандидатов – стохастический. Для одних и тех же входных параметров постоянно будут генерироваться различные результаты. Перед применением модели в приложении доступна только проверка работоспособности и отказоустойчивости отдельных компонентов с использованием модульного тестирования.

Однако, для проверки результативности модели, можно воспользоваться методами маркетинговых исследований. Наиболее популярным вариантом

является применение A/B-тестирования. Далее приводится пример плана действий для тестирования платформы микроблоггинга, так как в реальных условиях продукт не был завершен.

A/B эксперимент – сравнение показателей эффективности приложения при применении разных моделей. Для того, чтобы провести этот вид тестирования, требуется определение таксономии рекомендательной системы [3]. Требуется постановка следующих понятий для информационной системы, в которой будет производиться исследование:

– KPI – ключевой показатель эффективности. Именно эту метрику пытается оптимизировать команда разработки внедрением рекомендательной системы,

– понятие конверсии – положительного действия пользователя, которое мы хотим видеть в системе чаще,

– методы сбора данных о действиях пользователя. Зная последовательность действий пользователя, можно определить его путь конверсии (действия, которые он совершил до того, как произошла конверсия),

– методы анализа данных о действиях пользователя. Включает в себя способы расчета ключевых показателей и визуализацию метрик,

– сравниваемые варианты.

Будем рассматривать эффективность рекомендательной системы применительно к платформе микроблоггинга.

В платформах микроблоггинга пользователи публикуют короткие тексты, изображения и видео. Кроме публикации новых записей, можно оценивать посты других пользователей, переопубликовать их на своей странице или разместить свой ответ/комментарий, который точно так же является публикацией (см. рисунок 24).

Часто в качестве конверсии может выступать любое из этих действий точно так же, как и все сразу. Более подходящим вариантом будет формирование шкалы важности этих действий и их суммирование.

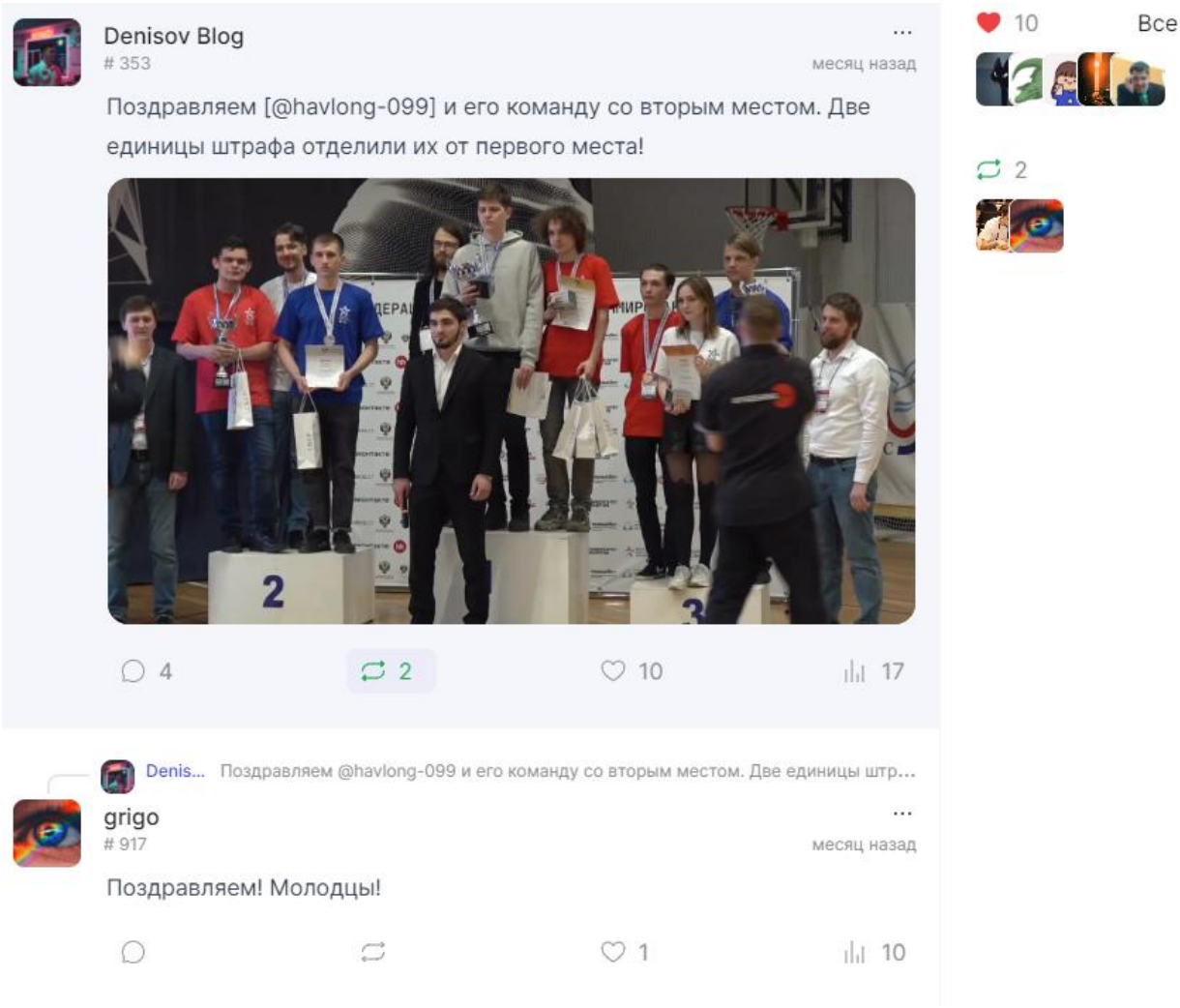


Рисунок 24 – Пример публикации на платформе микроблоггинга Вью

На платформе «Вью» в качестве оценки пользователем публикации выступает оценка «нравится».

Одной из наиболее универсальных метрик является CTR – показатель кликабельности, который рассчитывается как количество кликов по элементу относительно количества его просмотров. (см. рисунок 25)

$$\mathrm{CTR} = \frac{\text{Number of click-throughs}}{\text{Number of impressions}} \times 100(\%)$$

Рисунок 25 – Формула расчета кликабельности

Однако эта метрика не учитывает специфики различных приложений и намного более удобным показателем будет доля конверсий.

Введем понятие рейтинга – сумма конверсий для публикации. Тогда, если по модели Twitter, мы введем следующие коэффициенты для действий: *1.0* для

оценок «нравится», *2.0* для перепубликаций, *27.0* для ответов, то получим следующую формулу рейтинга:

$$rating = likes + 27 * replies + 2 * reposts$$

где $rating$ – рейтинг публикации,

$likes$ – количество лайков (оценок «нравится»),

$replies$ – количество ответных публикаций,

$reposts$ – количество перепубликаций.

В качестве KPI можно использовать суммарный рейтинг за период времени. Кроме того, стоит обратить внимание на этот показатель относительно количества просмотров (уникальных и неуникальных).

Оптимальным способом сбора данных будет изучение действий пользователя непосредственно в клиентском приложении. Для этого будем собирать как можно больше событий на странице персональных рекомендаций.

Введем следующие типы событий:

– VUE - пост из ленты находился на глазах у пользователя более 5 секунд;

– LONG_VUE - пост из ленты находился на глазах у пользователя более 15 секунд;

– NO_VUE - пост из ленты находился на глазах у пользователя более 2 минут;

– POST_OPENED - на пост перешли;

– POST_REPLIED - на пост открыли редактор ответов;

– POST_LIKE - пост понравился;

– POST_UNLIKE - пост больше не нравится;

– POST_REPUBLISH – пост переопубликовали на своей странице;

– POST_UNREPUBLISH - репост был отменен;

– POST_REPLY - на публикацию создан ответ;

– POST_SHARED - была нажата кнопка поделиться (скопировать ссылку);

– POST_CONTENT_ACTION - было сделано действие с содержимым;

1) картинка/видео раскрыты на полный экран;

2) был выбран вариант в опросе;

3) файл был скачан;

4) переход по внешней ссылке;

5) запуск аудио.

Кроме типа события будем передавать с запросами в базу данных аналитики его время, открытый раздел приложения, идентификатор пользователя и идентификатор публикации.

Для сравнения с рекомендательным движком будем применять систему, в которой элементы сортируются по относительному рейтингу и анализировать ее ключевые показатели эффективности.

3.5 Дальнейшие перспективы исследования

Текущее исследование показало успешные результаты в гибкости, компактности и скорости. Следующим шагом для развития рекомендательной системы является разработка полноценной экосистемы, включающей в себя фреймворк для визуализации собранной аналитики, интегрирование альтернативных источников кандидатов для рекомендации, ранкеров и миксеров.

За время разработки был произведен скачок и в альтернативных механизмах для формирования рекомендаций. Так, среди нейросетевых алгоритмов становятся более популярными обучение эмбеддингов графовых зависимостей [7], а также использование блоков, подобных механизмам внимания в архитектурах трансформеров, для извлечения признаков из данных [13, 14] о пользователях и кандидатах для рекомендации.

В дальнейшем планируется реализовать в рамках рекомендательной системы нейросетевой ранкер, добавить механизмы продвижения холодного контента и провести A/B тестирование, способное выявить наиболее эффективные конфигурации алгоритмов для различных информационных систем.

ЗАКЛЮЧЕНИЕ

В результате выполнения задания на магистерскую работу было разработано программное обеспечение рекомендательной системы на основе графовой модели.

Разработка проведена в системе Windows 10 с использованием языка программирования Kotlin в среде IntelliJ IDEA Community Edition.

В работе были рассмотрены основные принципы проектирования рекомендательных систем, разработан и оптимизирован графовый рекомендательный движок, позволяющий гибко встраивать решение в различные прикладные программы. Также, реализован механизм индексации графовых моделей и их распространения в виде файла или по сети. Для асинхронного исполнения запросов к рекомендательной системе использовалась технология Kotlin Coroutines.

В ходе выполнения работы были выполнены следующие задачи:

– изучение применяемых алгоритмов в области рекомендательных систем,

– анализ требований к разрабатываемой рекомендательной системе,

– составление концепции графовой рекомендательной системы,

– реализация рекомендательного движка на основе графовой модели,

– разработка механизма сериализации графовой модели,

– оптимизация скорости выполнения запросов к программному решению.

Осуществлено модульное тестирование разработанной программной библиотеки, которое показало корректность и эффективность его работы, а также соответствие поставленным требованиям.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Eksombatchai C. et al. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time // Proceedings of the 2018 world wide web conference. – 2018. – С. 1775-1784.
2. Sharma A. et al. GraphJet: Real-time content recommendations at Twitter // Proceedings of the VLDB Endowment. – 2016. – Т. 9. – №. 13. – С. 1281-1292.
3. Фальк К. Рекомендательные системы на практике // ДМК Пресс. – 2020.
4. Гришин, Е. В. Обзор алгоритмов и моделей рекомендательных систем / Е. В. Гришин, Л. В. Гурьянов // Информационные технологии в науке и образовании. Проблемы и перспективы: сборник статей по материалам X Всероссийской научно-практической конференции, Пенза, 15 марта 2023 года. – Пенза: Пензенский государственный университет, 2023. – С. 115-118.
5. Van den Oord A., Dieleman S., Schrauwen B. Deep content-based music recommendation //Advances in neural information processing systems. – 2013. – Т. 26.
6. Sarwar B. et al. Item-based collaborative filtering recommendation algorithms //Proceedings of the 10th international conference on World Wide Web. – 2001. – С. 285-295.
7. El-Kishky A. et al. TwHIN: Embedding the twitter heterogeneous information network for personalized recommendation //Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining. – 2022. – С. 2842-2850.
8. Lempel R., Moran S. SALSA: the stochastic approach for link-structure analysis //ACM Transactions on Information Systems (TOIS). – 2001. – Т. 19. – №. 2. – С. 131-160.
9. Gupta P. et al. Wtf: The who to follow service at twitter //Proceedings of the 22nd international conference on World Wide Web. – 2013. – С. 505-514.

10. Гришин, Е. В. Сериализация графовых моделей рекомендательных систем / Е. В. Гришин, Л. В. Гурьянов // Новые информационные технологии и системы (НИТиС-2022) : Сборник научных статей по материалам XIX Международной научно-технической конференции, посвященной 75-летию кафедры «Вычислительная техника» ПГУ, Пенза, 17–18 ноября 2022 года. – Пенза: Пензенский государственный университет, 2022. – С. 242-245.

11. Elizarov R. et al. Kotlin coroutines: design and implementation //Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. – 2021. – С. 68-84.

12. Гришин, Е. В. Применение Котлин сопрограмм в запросах к графовой рекомендательной системе / Е. В. Гришин, Л. В. Гурьянов // Новые информационные технологии и системы (НИТиС-2023) : Сборник научных статей по материалам XX Международной научно-технической конференции, посвященной 80-летнему юбилею Пензенского государственного университета, Пенза, 16–17 ноября 2023 года. – Пенза: Пензенский государственный университет, 2023. – С. 106-109.

13. Wang R. et al. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems //Proceedings of the web conference 2021. – 2021. – С. 1785-1797.

14. Wang Z., She Q., Zhang J. MaskNet: Introducing feature-wise multiplication to CTR ranking models by instance-guided mask //arXiv preprint arXiv:2102.07619. – 2021.

ПРИЛОЖЕНИЕ А

(рекомендуемое)

Исследование рекомендательных систем

на основе графовых моделей

Иллюстрационные материалы

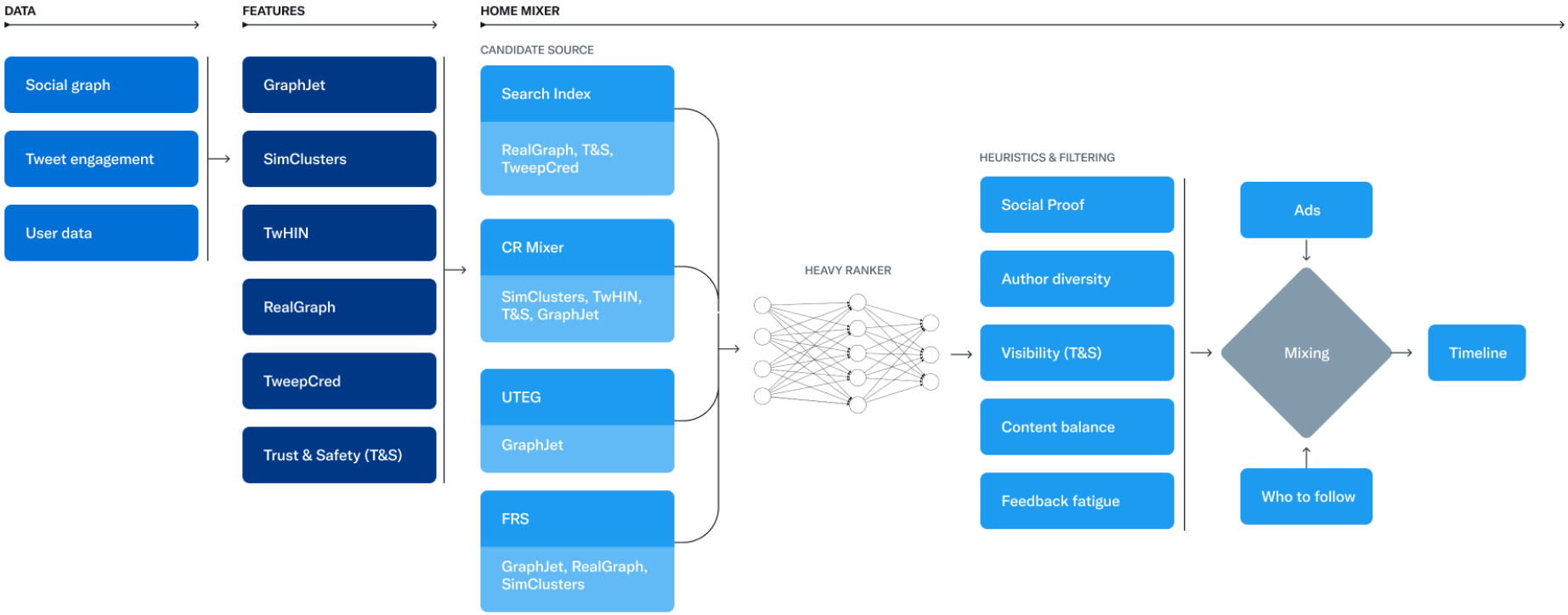


Рисунок А.1 – Архитектура рекомендательной системы Twitter

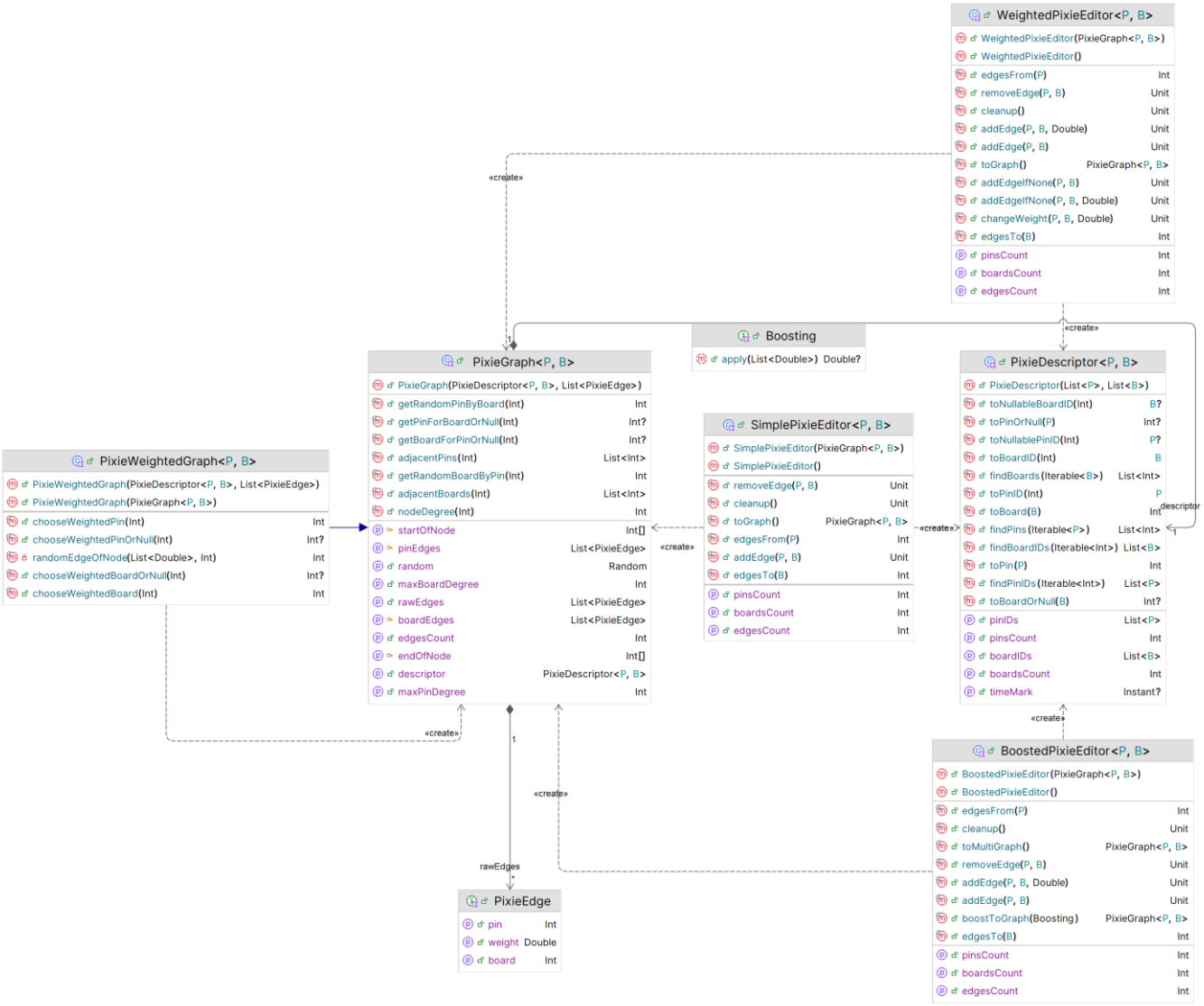


Рисунок А.2 – Диаграмма классов графовой модели рекомендательной систем

ПРИЛОЖЕНИЕ Б

(рекомендуемое)

Исследование рекомендательных систем

на основе графовых моделей

Листинг

PixieGraph.kt

```
/**
 * НЕИЗМЕНЯЕМЫЙ граф, используемый обработчиками. Для
редактирования рёбер и вершин используйте редакторы.
 * Используйте дескриптор [descriptor] для конвертирования
объектных идентификаторов в вершины графа.
 * [rawEdges] - сериализуемый список рёбер.
 */
@Serializable
open class PixieGraph<P, B>(
    val descriptor: PixieDescriptor<P, B>,
    val rawEdges: List<PixieEdge>
) {
    private val n = descriptor.pinsCount
    private val m = descriptor.boardsCount

    // индексы пинов
    @Transient
    private val edgesOfPin = 0 until n

    // индексы досок
    @Transient
    private val edgesOfBoard = n until n + m

    // Все рёбра вершин находятся в промежутке [startOfNode,
endOfNode].
    // endOfNode < startOfNode - нет рёбер, связанных с вершиной
    @Transient
    protected val startOfNode = IntArray(n + m)

    @Transient
    protected val endOfNode = IntArray(n + m)

    // Рёбра, отсортированные по пинам
    @Transient
    protected val pinEdges: List<PixieEdge> =
rawEdges.sortedWith(compareBy(PixieEdge::pin, PixieEdge::board))

    // Рёбра, отсортированные по доскам
    @Transient
    protected val boardEdges: List<PixieEdge> =
pinEdges.sortedWith(compareBy(PixieEdge::board, PixieEdge::pin))


    // Инициализируем startOfNode, endOfNode, pinEdges,
boardEdges
    init {
        var edge = 0
        for (node in edgesOfPin) {
            while (edge < pinEdges.size && pinEdges[edge].pin <
node) edge++
            startOfNode[node] = edge
```

```
            while (edge < pinEdges.size && pinEdges[edge].pin <=
node) edge++
            endOfNode[node] = edge - 1
        }

        edge = 0
        for (node in edgesOfBoard) {
            while (edge < boardEdges.size &&
boardEdges[edge].board < node) edge++
            startOfNode[node] = edge
            while (edge < boardEdges.size &&
boardEdges[edge].board <= node) edge++
            endOfNode[node] = edge - 1
        }
    }

    /**
     * Возвращает степень вершины [node].
     */
    fun nodeDegree(node: Int) = endOfNode[node] -
startOfNode[node] + 1

    // Максимальная степень пина
    @Transient
    val maxPinDegree = edgesOfPin.maxOfOrNull(::nodeDegree) ?: 0

    // Максимальная степень доски
    @Transient
    val maxBoardDegree = edgesOfBoard.maxOfOrNull(::nodeDegree)
?: 0

    // Генератор случайных чисел
    @Transient
    val random: Random =
ThreadLocalRandom.current().asKotlinRandom()

    /**
     * Возвращает случайную вершину доски, смежную с пином
[pin].
     */
    @Throws(IllegalArgumentException::class)
    fun getRandomBoardByPin(pin: Int): Int {
        val boardsAmount = nodeDegree(pin)
        if (boardsAmount <= 0) throw
IllegalArgumentException("Pin has no adjacent boards")
        val boardIndex = random.nextInt(boardsAmount)
        return pinEdges[startOfNode[pin] + boardIndex].board
    }

    /**
     * Возвращает случайную вершину доски, смежную с пином [pin]
или null.
     */
```

```
    fun getBoardForPinOrNull(pin: Int): Int? {
        val boardsAmount = nodeDegree(pin)
        if (boardsAmount <= 0) return null
        val boardIndex = random.nextInt(boardsAmount)
        return pinEdges[startOfNode[pin] + boardIndex].board
    }

    /**
     * Возвращает случайную вершину пина, смежную с доской
[board].
     */
    @Throws(IllegalArgumentException::class)
    fun getRandomPinByBoard(board: Int): Int {
        val pinsAmount = nodeDegree(board)
        if (pinsAmount <= 0) throw
IllegalArgumentException("Board has no adjacent pins")
        val pinIndex = random.nextInt(pinsAmount)
        return boardEdges[startOfNode[board] + pinIndex].pin
    }

    /**
     * Возвращает случайную вершину пина, смежную с доской
[board] или null.
     */
    fun getPinForBoardOrNull(board: Int): Int? {
        val pinsAmount = nodeDegree(board)
        if (pinsAmount <= 0) return null
        val pinIndex = random.nextInt(pinsAmount)
        return boardEdges[startOfNode[board] + pinIndex].pin
    }

    /**
     * Возвращает вершины досок, смежные с пином [pin].
     */
    fun adjacentBoards(pin: Int): List<Int> =
pinEdges.subList(startOfNode[pin], endOfNode[pin])
        .map(PixieEdge::board)

    /**
     * Возвращает вершины пинов, смежные с доской [board].
     */
    fun adjacentPins(board: Int): List<Int> =
boardEdges.subList(startOfNode[board], endOfNode[board])
        .map(PixieEdge::pin)

    /**
     * Возвращает количество рёбер в графе.
     */
    val edgesCount: Int
        get() = pinEdges.size
}
```

PixieWeightedGraph.kt

```
// Статические интегральные суммы хранятся в накапливаемых
массивах
typealias StaticSum = List<Double>

// Преобразование к массиву интегральных сумм
private fun StaticSum.toStaticSum(): StaticSum =
runningFold(0.0, Double::plus)

// Запрос суммы на отрезке [l, r)
private fun StaticSum.query(l: Int, r: Int): Double = this[r] -
this[l]

// Поиск значения value на отрезке [from, to) с использованием
бинарного поиска
private fun StaticSum.search(from: Int, to: Int, value: Double):
Int {
    // Функция проверяющая значение на валидность, искомое
значение должно быть меньше или равно искомому
    val valid: (Int) -> Boolean = { i -> value >= query(from, i)
}

    // Границы бинарного поиска
    var l = from
    var r = to
    // Пока между границами есть хотя бы один элемент
    while (r > l + 1) {
        val mid = (l + r) shr 1
        if (valid(mid)) {
            l = mid
        } else {
            r = mid
        }
    }

    // Проверем правую границу перед возвратом. При наличии
невалидного значения возвращаем левую границу
    return if (valid(r)) r else l
}

/**
 * Граф, применяющий кеширование весов для быстрых запросов к
случайным вершинам с учетом весов рёбер.
 */
class PixieWeightedGraph<P, B>(
    descriptor: PixieDescriptor<P, B>,
    rawEdges: List<PixieEdge>
) : PixieGraph<P, B>(descriptor, rawEdges) {

    constructor(from: PixieGraph<P, B>) : this(from.descriptor,
from.rawEdges)
```

```
    // Массив интегральных сумм для пинов
    @Transient
    private val cachedPinWeights: StaticSum =
pinEdges.map(PixieEdge::weight).toStaticSum()

    // Массив интегральных сумм для досок
    @Transient
    private val cachedBoardWeights: StaticSum =
boardEdges.map(PixieEdge::weight).toStaticSum()

    /**
     * Возвращает случайную вершину доски, смежную с пином
[pin].
     * Расчет производится с учетом весов рёбер.
     */
    @Throws(IllegalArgumentException::class)
    fun chooseWeightedBoard(pin: Int): Int {
        val chosenBoardEdge = randomEdgeOfNode(cachedPinWeights,
pin)
            ?: throw IllegalArgumentException("Pin has no
adjacent boards")

        return pinEdges[chosenBoardEdge].board
    }

    /**
     * Возвращает случайную вершину доски, смежную с пином [pin]
или null.
     * Расчет производится с учетом весов рёбер.
     */
    fun chooseWeightedBoardOrNull(pin: Int): Int? =
        randomEdgeOfNode(cachedPinWeights, pin)?.let {
pinEdges[it].board }

    /**
     * Возвращает случайную вершину пина, смежную с доской
[board].
     * Расчет производится с учетом весов рёбер.
     */
    @Throws(IllegalArgumentException::class)
    fun chooseWeightedPin(board: Int): Int {
        val chosenPinEdge = randomEdgeOfNode(cachedBoardWeights,
board)
            ?: throw IllegalArgumentException("Board has no
adjacent pins")

        return boardEdges[chosenPinEdge].pin
    }

    /**
     * Возвращает случайную вершину пина, смежную с доской
[board] или null.
     * Расчет производится с учетом весов рёбер.
```

```
     */
    fun chooseWeightedPinOrNull(board: Int): Int? =
        randomEdgeOfNode(cachedBoardWeights, board)?.let {
boardEdges[it].pin }

    /**
     * Возвращает случайное ребро, смежное с вершиной графа
[node].
     * Расчет производится с учетом весов рёбер.
     * Список рёбер [cachedPart] должен быть отсортирован в
соответствии с долей вершины.
     */
    private fun randomEdgeOfNode(cachedPart: StaticSum, node:
Int): Int? {
        if (nodeDegree(node) <= 0) return null

        val sum = cachedPart.query(startOfNode[node],
endOfNode[node] + 1)

        val chosen = random.nextDouble(sum)
        val edgeIndex = cachedPart.search(startOfNode[node],
endOfNode[node] + 1, chosen)

        assert(edgeIndex in startOfNode[node]..endOfNode[node])
        return edgeIndex
    }
}
```

PixieEdge.kt

```
interface PixieEdge {
    val pin: Int
    val board: Int
    val weight: Double
}
```

PixieEdgeSimple.kt

```
@Serializable
@SerialName("eS")
data class PixieEdgeSimple(
    @SerialName("l") override val pin: Int,
    @SerialName("r") override val board: Int
) : PixieEdge {
    override val weight: Double
        get() = DEFAULT_WEIGHT
}
```

PixieEdgeWeighted.kt

```
@Serializable
@SerialName("eW")
data class PixieEdgeWeighted(
```

```
    @SerialName("l") override val pin: Int,
    @SerialName("r") override val board: Int,
    @SerialName("w") override val weight: Double =
DEFAULT_WEIGHT
) : PixieEdge
```

PixieDescriptor.kt

```
/**
 * Дескриптор графа с информацией, необходимой для взаимной
конвертации вершин графа и объектных идентификаторов.
 */
@Serializable
class PixieDescriptor<P, B>(
    val pinIDs: List<P>,
    val boardIDs: List<B>
) {

    /**
     * Метка времени. Временно не используется и равна null.
     */
    @Serializable(with = InstantSerializer::class)
    @SerialName("timeMark")
    var timeMark: Instant? = Instant.now()

    // Конвертеры объектных идентификаторов в вершины графа
    @Transient
    private val pins: Map<P, Int> = buildMap {
        pinIDs.forEachIndexed { index, pin -> this[pin] = index
}
    }

    @Transient
    private val boards: Map<B, Int> = buildMap {
        boardIDs.forEachIndexed { index, board -> this[board] =
index + pinIDs.size }
    }

    /**
     * Количество пинов в графе.
     */
    val pinsCount: Int
        get() = pinIDs.size

    /**
     * Количество досок в графе.
     */
    val boardsCount: Int
        get() = boardIDs.size

    /**
     * Конвертирует вершину графа в объектный идентификатор
пина.
```

```
     */
    fun toPinID(pin: Int): P = pinIDs[pin]

    /**
     * Конвертирует вершину графа в объектный идентификатор
доски.
     */
    fun toBoardID(board: Int): B = boardIDs[board - pinsCount]

    /**
     * Конвертирует пин в вершину.
     */
    fun toPin(pinID: P): Int = pins[pinID]!!

    /**
     * Конвертирует пин в вершину.
     */
    fun toBoard(boardID: B): Int = boards[boardID]!!


    /**
     * Конвертирует вершину графа в объектный идентификатор пина
или возвращает null.
     */
    fun toNullablePinID(pin: Int): P? = pin.takeIf { it in
pinIDs.indices }?.let(::toPinID)

    /**
     * Конвертирует вершину графа в объектный идентификатор
доски или возвращает null.
     */
    fun toNullableBoardID(board: Int): B? = board.takeIf { (it -
pinsCount) in boardIDs.indices }?.let(::toBoardID)

    /**
     * Конвертирует пин в вершину, или возвращает null.
     */
    fun toPinOrNull(pinID: P): Int? = pins[pinID]

    /**
     * Конвертирует пин в вершину, или возвращает null.
     */
    fun toBoardOrNull(boardID: B): Int? = boards[boardID]


    /**
     * Конвертирует список вершин в список объектных
идентификаторов пинов.
     */
    fun Iterable<Int>.findPinIDs(): List<P> =
mapNotNull(::toNullablePinID)

    /**
```

```
     * Конвертирует список вершин в список объектных
идентификаторов досок.
     */
    fun Iterable<Int>.findBoardIDs(): List<B> =
mapNotNull(::toNullableBoardID)

    /**
     * Конвертирует список пинов в список вершин графа.
     *
     * Если пина не было, то он пропускается
     */
    fun Iterable<P>.findPins(): List<Int> =
mapNotNull(::toPinOrNull)

    /**
     * Конвертирует список досок в список вершин графа.
     *
     * Если доски не было, то он пропускается
     */
    fun Iterable<B>.findBoards(): List<Int> =
mapNotNull(::toBoardOrNull)
}
```

PixieQueryManager.kt

```
class PixieQueryManager(
    queryThreads: Int,
    private val pixieWalkerStorage: PixieWalkerStorage
) {
    private val pixieThreadPool =
Executors.newFixedThreadPool(queryThreads)
    private val pixieDispatcher =
pixieThreadPool.asCoroutineDispatcher()
    private val pixieScope = CoroutineScope(pixieDispatcher +
CoroutineName("Pixie Query"))

    /**
     * Производит простую случайную прогулку по графу и
возвращает [PixieWalker], содержащий результаты.
     * Обход производится по графу [graph], начиная из пина
[queryPinID], выполняя не более [maxSteps] шагов.
     * Параметр [alpha] - вероятность вернуться в начало.
     */
    fun <P, B> walkSimple(
        graph: PixieGraph<P, B>,
        queryPinID: P,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        // Конвертируем пин запроса в вершину графа
        val qPin = queryPinID.asPinOf(graph) { return@async
EmptyWalker }
```

```
        pixieWalkerStorage.lockVisitWalker(graph).apply {
            walk(
                steps = maxSteps,
                pinProvider = graph::getRandomPinByBoard,
                boardProvider = graph::getRandomBoardByPin,
                shouldReset = { graph.random.nextDouble() <
alpha },
                reset = { qPin }
            )
        }
    }

    /**
     * Производит простую случайную прогулку по графу и
возвращает [PixieWalker], содержащий результаты.
     * Обход производится по графу [graph], начиная из доски
[queryBoardID], выполняя не более [maxSteps] шагов.
     * Параметр [alpha] - вероятность вернуться в начало.
     */
    fun <P, B> walkSimpleBoard(
        graph: PixieGraph<P, B>,
        queryBoardID: B,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        // Конвертируем доску запроса в вершину графа
        val qBoard = queryBoardID.asBoardOf(graph) {
return@async EmptyWalker }

        pixieWalkerStorage.lockVisitWalker(graph).apply {
            walk(
                steps = maxSteps,
                pinProvider = graph::getRandomPinByBoard,
                boardProvider = graph::getRandomBoardByPin,
                shouldReset = { graph.random.nextDouble() <
alpha },
                reset = { graph.getRandomPinByBoard(qBoard) }
            )
        }
    }

    /**
     * Производит прогулку по графу без учёта весов рёбер и
возвращает [PixieWalker].
     * При посещении [peakPinsToChoose] пинов не менее
[peakPointToAchieve] раз останавливает работу предварительно.
     * Для понимания концепции прогулки обратитесь к
[walkSimple].
     */
    fun <P, B> walkSimpleWithStop(
        graph: PixieGraph<P, B>,
```

```
        queryPinID: P,
        maxSteps: Int,
        alpha: Double,
        peakPinsToChoose: Int,
        peakPointToAchieve: Int
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        // Конвертируем пин запроса в вершину графа
        val qPin = queryPinID.asPinOf(graph) { return@async
EmptyWalker }

        pixieWalkerStorage.lockVisitWalker(graph).apply {
            walkWithStop(
                steps = maxSteps,
                threshold = peakPointToAchieve,
                stopOnAmount = peakPinsToChoose,
                pinProvider = graph::getRandomPinByBoard,
                boardProvider = graph::getRandomBoardByPin,
                shouldReset = { graph.random.nextDouble() <
alpha },
                reset = { qPin }
            )
        }
    }

    /**
     * Производит прогулку по графу без учёта весов рёбер и
возвращает [PixieWalker].
     * При посещении [peakPinsToChoose] пинов не менее
[peakPointToAchieve] раз останавливает работу предварительно.
     * Для понимания концепции прогулки обратитесь к
[walkSimpleBoard].
     */
    fun <P, B> walkSimpleBoardWithStop(
        graph: PixieGraph<P, B>,
        queryBoardID: B,
        maxSteps: Int,
        alpha: Double,
        peakPinsToChoose: Int,
        peakPointToAchieve: Int
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        // Конвертируем доску запроса в вершину графа
        val qBoard = queryBoardID.asBoardOf(graph) {
return@async EmptyWalker }

        pixieWalkerStorage.lockVisitWalker(graph).apply {
            walkWithStop(
                steps = maxSteps,
                threshold = peakPointToAchieve,
                stopOnAmount = peakPinsToChoose,
                pinProvider = graph::getRandomPinByBoard,
                boardProvider = graph::getRandomBoardByPin,
```

```
                shouldReset = { graph.random.nextDouble() <
alpha },
                reset = { graph.getRandomPinByBoard(qBoard) }
            )
        }
    }

    /**
     * Производит прогулку по графу с возможностью старта из
нескольких вершин [queriedPins] вместе с их весами,
     * обозначающими их важность. [walkSimple] стартует из
каждой вершины с пропорциональным количеством шагов.
     * Для объединения результатов обхода используется
[BoostedPixieWalker], возвращаемый этим методом.
     * Для понимания концепции прогулки обратитесь к
[walkSimple].
     */
    fun <P, B> walkMultiSimple(
        graph: PixieGraph<P, B>,
        queriedPins: List<Weighted<P>>,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieConsumeWalker> = pixieScope.async {

        // Распределяем шаги между запрашиваемыми вершинами в
соответствии с весами и степенями
        val scaledPins = scaleStepsByPins(maxSteps, graph,
queriedPins)

        // Захватываем Бустер, который будет объединять
результаты различных обходов
        pixieWalkerStorage.lockConsumeWalker(graph).also {
consumeWalker ->
            channelFlow {
                scaledPins.forEach { (pin, steps) ->
                    // Массовый запуск обходов
                    launch {
                        // Пытаемся захватить из пула обходчик
для случайной прогулки
                        val walker =
pixieWalkerStorage.lockVisitWalker(graph)
                        walker.walk(
                            steps = steps,
                            pinProvider =
graph::getRandomPinByBoard,
                            boardProvider =
graph::getRandomBoardByPin,
                            shouldReset = {
graph.random.nextDouble() < alpha },
                            reset = { pin }
                        )
                        // Отправляем обходчик в поток
результатов
```

```
                        send(walker)
                    }
                }
            }
                .onEach(consumeWalker::consume)
                .onEach(pixieWalkerStorage::releaseVisitWalker)
// Обязательно возвращаем обходчик в пул
                .collect()
        }
    }

    /**
     * Производит прогулку по графу с возможностью старта из
нескольких вершин-досок [queriedBoards] вместе с их весами,
     * обозначающими их важность. [walkSimpleBoard] стартует из
каждой вершины с пропорциональным количеством шагов.
     * Для объединения результатов обхода используется
[BoostedPixieWalker], возвращаемый этим методом.
     * Для понимания концепции прогулки обратитесь к
[walkSimpleBoard].
     */
    fun <P, B> walkMultiSimpleBoards(
        graph: PixieGraph<P, B>,
        queriedBoards: List<Weighted<B>>,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieConsumeWalker> = pixieScope.async {

        val scaledBoards = scaleStepsByBoards(maxSteps, graph,
queriedBoards)

        pixieWalkerStorage.lockConsumeWalker(graph).also {
consumeWalker ->
            channelFlow {
                scaledBoards.forEach { (board, steps) ->
                    launch {
                        val walker =
pixieWalkerStorage.lockVisitWalker(graph)
                        walker.walk(
                            steps = steps,
                            pinProvider =
graph::getRandomPinByBoard,
                            boardProvider =
graph::getRandomBoardByPin,
                            shouldReset = {
graph.random.nextDouble() < alpha },
                            reset = {
graph.getRandomPinByBoard(board) }
                        )
                        send(walker)
                    }
                }
            }
```

```
                .onEach(consumeWalker::consume)
                .onEach(pixieWalkerStorage::releaseVisitWalker)
                .collect()
        }
    }

    /**
     * Производит прогулку по графу с учётом весов рёбер и
возвращает [PixieWalker].
     * Для понимания концепции прогулки обратитесь к
[walkSimple].
     */
    fun <P, B> walkWeighted(
        weightedGraph: PixieWeightedGraph<P, B>,
        queryPinID: P,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        val qPin = queryPinID.asPinOf(weightedGraph) {
return@async EmptyWalker }

        pixieWalkerStorage.lockVisitWalker(weightedGraph).apply
{
            walk(
                steps = maxSteps,
                pinProvider = weightedGraph::chooseWeightedPin,
                boardProvider =
weightedGraph::chooseWeightedBoard,
                shouldReset = {
weightedGraph.random.nextDouble() < alpha },
                reset = { qPin }
            )
        }
    }

    /**
     * Производит прогулку по графу с учётом весов рёбер и
возвращает [PixieWalker].
     * Для понимания концепции прогулки обратитесь к
[walkSimpleBoard].
     */
    fun <P, B> walkWeightedBoard(
        weightedGraph: PixieWeightedGraph<P, B>,
        queryBoardID: B,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        val qBoard = queryBoardID.asBoardOf(weightedGraph) {
return@async EmptyWalker }
```

```
        pixieWalkerStorage.lockVisitWalker(weightedGraph).apply
{
            walk(
                steps = maxSteps,
                pinProvider = weightedGraph::chooseWeightedPin,
                boardProvider =
weightedGraph::chooseWeightedBoard,
                shouldReset = {
weightedGraph.random.nextDouble() < alpha },
                reset = {
weightedGraph.chooseWeightedPin(qBoard) }
            )
        }
    }

    /**
     * Производит прогулку по графу с учётом весов рёбер и
возвращает [PixieWalker].
     * При посещении [peakPinsToChoose] пинов не менее
[peakPointToAchieve] раз останавливает работу предварительно.
     * Для понимания концепции прогулки обратитесь к
[walkSimple].
     */
    fun <P, B> walkWeightedWithStop(
        weightedGraph: PixieWeightedGraph<P, B>,
        queryPinID: P,
        maxSteps: Int,
        alpha: Double,
        peakPinsToChoose: Int,
        peakPointToAchieve: Int
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        val qPin = queryPinID.asPinOf(weightedGraph) {
return@async EmptyWalker }

        pixieWalkerStorage.lockVisitWalker(weightedGraph).apply
{
            walkWithStop(
                steps = maxSteps,
                threshold = peakPointToAchieve,
                stopOnAmount = peakPinsToChoose,
                pinProvider = weightedGraph::chooseWeightedPin,
                boardProvider =
weightedGraph::chooseWeightedBoard,
                shouldReset = {
weightedGraph.random.nextDouble() < alpha },
                reset = { qPin }
            )
        }
    }

    /**
```

```
     * Производит прогулку по графу с учётом весов рёбер и
возвращает [PixieWalker].
     * При посещении [peakPinsToChoose] пинов не менее
[peakPointToAchieve] раз останавливает работу предварительно.
     * Для понимания концепции прогулки обратитесь к
[walkSimple].
     */
    fun <P, B> walkWeightedBoardWithStop(
        weightedGraph: PixieWeightedGraph<P, B>,
        queryBoardID: B,
        maxSteps: Int,
        alpha: Double,
        peakPinsToChoose: Int,
        peakPointToAchieve: Int
    ): Deferred<PixieVisitWalker> = pixieScope.async {

        val qBoard = queryBoardID.asBoardOf(weightedGraph) {
return@async EmptyWalker }

        pixieWalkerStorage.lockVisitWalker(weightedGraph).apply
{
            walkWithStop(
                steps = maxSteps,
                threshold = peakPointToAchieve,
                stopOnAmount = peakPinsToChoose,
                pinProvider = weightedGraph::chooseWeightedPin,
                boardProvider =
weightedGraph::chooseWeightedBoard,
                shouldReset = {
weightedGraph.random.nextDouble() < alpha },
                reset = {
weightedGraph.chooseWeightedPin(qBoard) }
            )
        }
    }

    /**
     * Производит прогулку по графу с возможностью старта из
нескольких вершин [queriedPins] вместе с их весами,
     * обозначающими их важность. [walkWeighted] стартует из
каждой вершины с пропорциональным количеством шагов.
     * Для объединения результатов обхода используется
[BoostedPixieWalker], возвращаемый этим методом.
     * Для понимания концепции прогулки обратитесь к
[walkSimple].
     */
    fun <P, B> walkMultiWeighted(
        weightedGraph: PixieWeightedGraph<P, B>,
        queriedPins: List<Weighted<P>>,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieConsumeWalker> = pixieScope.async {
```

```
        val scaledPins = scaleStepsByPins(maxSteps,
weightedGraph, queriedPins)

        pixieWalkerStorage.lockConsumeWalker(weightedGraph).also
{ consumeWalker ->
            channelFlow {
                scaledPins.forEach { (pin, steps) ->
                    launch {
                        val walker =
pixieWalkerStorage.lockVisitWalker(weightedGraph)
                        walker.walk(
                            steps = steps,
                            pinProvider =
weightedGraph::chooseWeightedPin,
                            boardProvider =
weightedGraph::chooseWeightedBoard,
                            shouldReset = {
weightedGraph.random.nextDouble() < alpha },
                            reset = { pin }
                        )
                        send(walker)
                    }
                }
            }
                .onEach(consumeWalker::consume)
                .onEach(pixieWalkerStorage::releaseVisitWalker)
                .collect()
        }
    }

    /**
     * Производит прогулку по графу с возможностью старта из
нескольких вершин-досок [queriedBoards] вместе с их весами,
     * обозначающими их важность. [walkWeightedBoard] стартует
из каждой вершины с пропорциональным количеством шагов.
     * Для объединения результатов обхода используется
[BoostedPixieWalker], возвращаемый этим методом.
     * Для понимания концепции прогулки обратитесь к
[walkSimpleBoard].
     */
    fun <P, B> walkMultiWeightedBoards(
        weightedGraph: PixieWeightedGraph<P, B>,
        queriedBoards: List<Weighted<B>>,
        maxSteps: Int,
        alpha: Double
    ): Deferred<PixieConsumeWalker> = pixieScope.async {

        val scaledBoards = scaleStepsByBoards(maxSteps,
weightedGraph, queriedBoards)

        pixieWalkerStorage.lockConsumeWalker(weightedGraph).also
{ consumeWalker ->
            channelFlow {
```

```
                scaledBoards.forEach { (board, steps) ->
                    launch {
                        val walker =
pixieWalkerStorage.lockVisitWalker(weightedGraph)
                        walker.walk(
                            steps = steps,
                            pinProvider =
weightedGraph::chooseWeightedPin,
                            boardProvider =
weightedGraph::chooseWeightedBoard,
                            shouldReset = {
weightedGraph.random.nextDouble() < alpha },
                            reset = {
weightedGraph.chooseWeightedPin(board) }
                        )
                        send(walker)
                    }
                }
            }
                .onEach(consumeWalker::consume)
                .onEach(pixieWalkerStorage::releaseVisitWalker)
                .collect()
        }
    }

}
```

ПРИЛОЖЕНИЕ В

(рекомендуемое)

Исследование рекомендательных систем

на основе графовых моделей

Презентация

Исследование рекомендательных систем на основе графовых моделей

Подготовил:
Студент группы 22ВВИм2 Гришин Евгений Валентинович
Руководитель:
Доцент кафедры МОиПЭВМ Гурьянов Лев Вячеславович

Пенза 2024

Цель и задачи

- Объект исследования – рекомендательные системы
- Предмет исследования – графовые рекомендательные системы
- Цель исследования – разработка рекомендательной системы, функционирующей в условиях ограниченных ресурсов
- Задачи исследования:
  - Провести анализ применяемых алгоритмов в области рекомендательных систем
  - Составить концепцию графовой рекомендательной системы
  - Оптимизировать программное решение для выполнения технических требований
  - Произвести тестирование и внедрение разработанных модулей

Актуальность

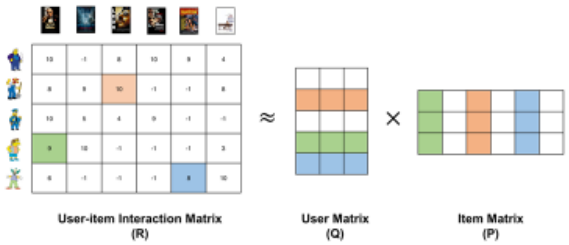
- Рекомендательные системы увеличивают эффективность приложений, взаимодействующих с людьми, добавляя элемент персонализации
- Графовые модели рекомендательных систем позволяют сохранять структуру предметной области, сокращая расходы на вычисления
- Предлагаемое решение показало гибкость при адаптации системы к различным условиям по сравнению с аналогами

3

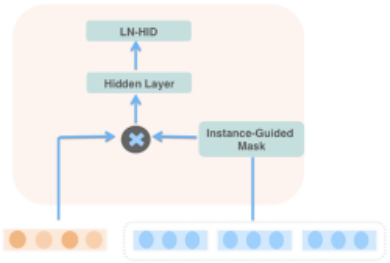
Кратко о рекомендательных системах: классификация

4

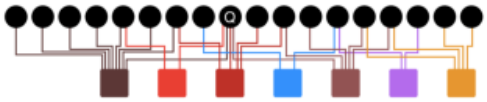
Кратко о рекомендательных системах: алгоритмы



классические



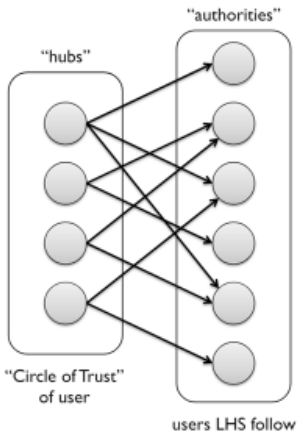
нейросетевые



графовые

5

Графовые решения: GraphJet



Плюсы:

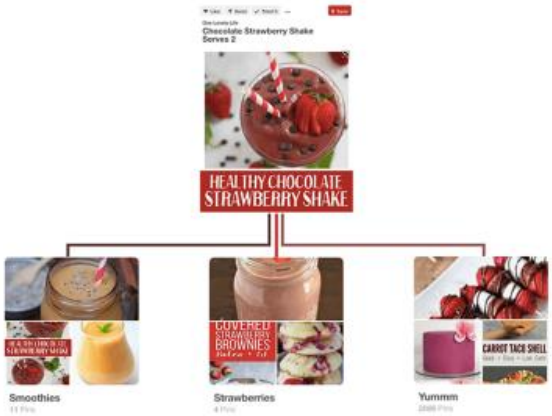
- Простота
- Скорость
- Совместимость с JVM

Минусы:

- Гомогенность
- Ограниченная индексация

6

Графовые решения: Pixie



Плюсы:

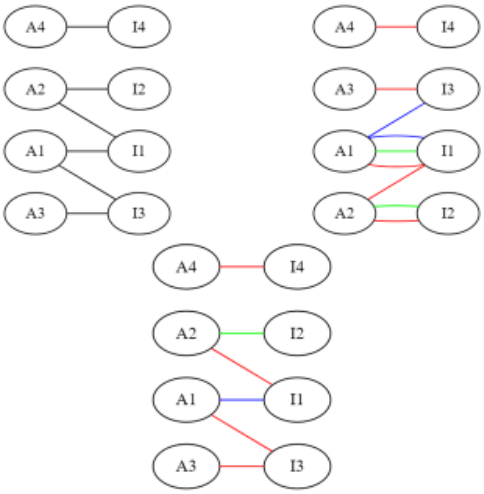
- Повышенная персонализация
- Высокая скорость вычислений

Минусы:

- Гомогенность
- Эвристический профиль пользователя
- Маршаллинг из C++

7

Графовые решения: PhoenixRS



Плюсы:

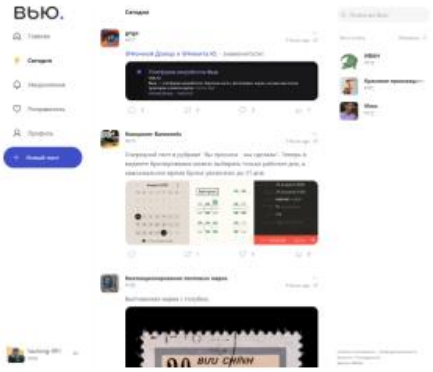
- Гибкость модели
- Гибкость индексации
- Гетерогенность
- Совместимость с JVM

Минусы:

- Скорость ниже аналогов
- Требует зависимого кода

8

Способность к адаптации



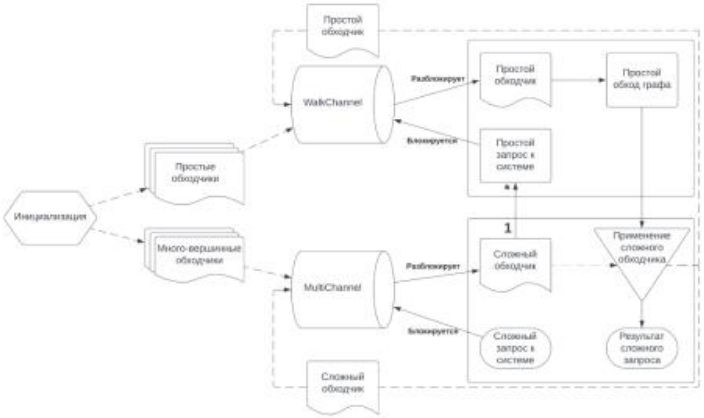
Vue Today – раздел персонализированных рекомендаций на платформе Вью

Похожие арты в приложении раскрасок Sandbox

Секция «Ещё арты» на платформе пиксель-арта Stich Pixel Art

9

Параллелизация запросов



10

Тестирование вычислительных возможностей

Количество потоков в пуле менеджера	1 поток	2 потока	4 потока	6 потоков
Корректность суммы итераций (рейтингов)	да	да	да	да
Время простых запросов	184 мс	141 мс	97 мс	113 мс
Время простых запросов без извлечения рейтингов	162 мс	92 мс	71 мс	75 мс

11

Подведение итогов исследования

- Система обеспечивает требуемую вычислительную эффективность
- Прототип рекомендательной системы был гибко интегрирован в несколько решений
- Существует возможность дальнейшего развития экосистемы рекомендательной системы PhoenixRS
- Материалы исследования опубликованы в цикле научных статей

12