* Вопросы не подразумевают единственно верного варианта решения задачи и служат прежде всего для калибровки навыка мышления в продуктово значимой парадигме
* Вопрос может не быть напрямую связан с технологией, если явно не указано.

**Вопросы:**

1. Вопрос связан с архитектурой построения таблиц БД и обвязки в виде Кеша. В ответе указать не просто схему организации, но объяснить ход мышления.

Дано: есть бизнес структура. Она состоит из товаров которые продаются в разных ГЕО. При этом в каждом ГЕО есть своя валюта и своя цена (учитывая доставку). К каждому ГЕО, на каждый товар приходят лиды, которых надо фиксировать. При этом есть коэффициент который динамически повышает или понижает цену в зависимости от количества лидов пришедших на ГЕО и товар. По принципу - больше лидов - меньше цена. Рейтинг обновляется раз в 10 минут.

Отразите схему организации мест хранения данных и взаимодействия с ними в виде запросов.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Для данной задачи схема организации мест хранения данных и взаимодействия с ними в виде запросов может выглядеть следующим образом. Мы разобьем задачу на несколько основных сущностей и их связи, а затем определим, какие запросы могут быть использованы для их взаимодействия. 1. Сущности:  * **Товары (Products)** * **ГЕО (Geos)** * **Лиды (Leads)** * **Цены (Prices)** * **Коэффициенты (Coefficients)**  2. Структура таблиц (Сущности)2.1. Таблица products Содержит информацию о товарах.   |  |  |  | | --- | --- | --- | | Поле | Тип | Описание | | product\_id | INT | Уникальный идентификатор товара | | product\_name | VARCHAR(255) | Название товара |  2.2. Таблица geos Содержит информацию о географических регионах (ГЕО).   |  |  |  | | --- | --- | --- | | Поле | Тип | Описание | | geo\_id | INT | Уникальный идентификатор ГЕО | | geo\_name | VARCHAR(255) | Название региона | | currency | VARCHAR(10) | Валюта региона |  2.3. Таблица prices Содержит информацию о ценах для товаров в разных ГЕО.   |  |  |  | | --- | --- | --- | | Поле | Тип | Описание | | price\_id | INT | Уникальный идентификатор цены | | product\_id | INT | Идентификатор товара | | geo\_id | INT | Идентификатор ГЕО | | price | DECIMAL(10,2) | Цена товара в выбранной валюте | | delivery\_cost | DECIMAL(10,2) | Стоимость доставки в данном ГЕО | | total\_price | DECIMAL(10,2) | Полная цена товара с учетом доставки |  2.4. Таблица leads Содержит информацию о лидах для товаров в различных ГЕО.   |  |  |  | | --- | --- | --- | | Поле | Тип | Описание | | lead\_id | INT | Уникальный идентификатор лида | | product\_id | INT | Идентификатор товара | | geo\_id | INT | Идентификатор ГЕО | | lead\_count | INT | Количество лида на товар в данном ГЕО | | timestamp | TIMESTAMP | Время получения лида |  2.5. Таблица coefficients Содержит коэффициенты для расчета динамической цены на основе лидов.   |  |  |  | | --- | --- | --- | | Поле | Тип | Описание | | product\_id | INT | Идентификатор товара | | geo\_id | INT | Идентификатор ГЕО | | lead\_threshold | INT | Пороговое количество лидов для изменения цены | | coefficient | DECIMAL(5,3) | Коэффициент для изменения цены |  3. Запросы3.1. Запрос на обновление цены товара в ГЕО (в зависимости от лидов) Для того чтобы обновить цену товара в зависимости от количества лидов, нужно выполнить следующие шаги:   1. Рассчитываем коэффициент на основе текущего количества лидов для товара в определенном ГЕО. 2. Применяем коэффициент к базовой цене товара с учетом доставки.   Пример запроса для обновления цены товара:   |  | | --- | | sql | | WITH lead\_summary AS (  SELECT product\_id, geo\_id, SUM(lead\_count) AS total\_leads  FROM leads  WHERE timestamp > NOW() - INTERVAL 10 MINUTE  GROUP BY product\_id, geo\_id  ),  updated\_prices AS (  SELECT p.product\_id, p.geo\_id, pr.price, pr.delivery\_cost,  ls.total\_leads,  COALESCE(c.coefficient, 1) AS coefficient  FROM lead\_summary ls  JOIN prices pr ON pr.product\_id = ls.product\_id AND pr.geo\_id = ls.geo\_id  LEFT JOIN coefficients c ON c.product\_id = pr.product\_id AND c.geo\_id = pr.geo\_id  )  UPDATE prices pr  JOIN updated\_prices up ON pr.product\_id = up.product\_id AND pr.geo\_id = up.geo\_id  SET pr.price = (up.price \* (1 - up.coefficient)) + up.delivery\_cost  WHERE pr.product\_id = up.product\_id AND pr.geo\_id = up.geo\_id; |  3.2. Запрос на получение текущего количества лидов для товара в ГЕО за последние 10 минут Этот запрос позволяет получить количество лидов для товара в ГЕО за последние 10 минут:   |  | | --- | | sql | | SELECT product\_id, geo\_id, SUM(lead\_count) AS total\_leads  FROM leads  WHERE timestamp > NOW() - INTERVAL 10 MINUTE  GROUP BY product\_id, geo\_id; |  3.3. Запрос на получение текущей цены товара для конкретного ГЕО Для получения актуальной цены товара с учетом доставки и коэффициента:   |  | | --- | | sql | | SELECT pr.product\_id, pr.geo\_id, pr.price, pr.delivery\_cost, pr.total\_price  FROM prices pr  WHERE pr.product\_id = ? AND pr.geo\_id = ?; |  3.4. Запрос на обновление коэффициента для товара в ГЕО Этот запрос используется, если нужно обновить коэффициент для расчета цены товара на основе количества лидов:   |  | | --- | | sql | | UPDATE coefficients  SET coefficient = ?  WHERE product\_id = ? AND geo\_id = ?; |  3.5. Запрос на получение всех товаров с их актуальными ценами в разных ГЕО  |  | | --- | | sql | | SELECT p.product\_name, g.geo\_name, pr.price, pr.delivery\_cost, pr.total\_price  FROM products p  JOIN prices pr ON p.product\_id = pr.product\_id  JOIN geos g ON pr.geo\_id = g.geo\_id  WHERE pr.product\_id = ?; |  4. Интеграция и периодическое обновление Для реализации динамического пересчета цены каждые 10 минут можно настроить задачу в планировщике (например, с помощью cron на Linux или аналогичного инструмента), который будет запускать обновление цен с указанной частотой. |

1. Вопрос по организации кода.

Необходимо сделать сервис для лайк/дизлайк, который может быть интегрирован в разные места проекта, учитывающий что на разных страницах есть разные сущности для лайков. Количество экшнов за день равно от 1млн. Опишите с точки зрения технологии, языков и базовой структуры как вы организуете код и взаимодействие его частей друг с другом

|  |
| --- |
| 1. **Общие требования и ограничения:**  * **Масштабируемость:** Система должна обрабатывать до 1 миллиона экшнов в день, что требует высокой производительности, устойчивости к нагрузке и быстрого отклика. * **Гибкость:** Логика лайков должна быть интегрируемой в различные страницы проекта, для разных сущностей (пользователи, статьи, комментарии, изображения и т.д.). * **Независимость:** Каждый тип сущности (например, лайки для постов, лайки для комментариев) должен быть изолирован и иметь свою логику, но при этом сервис должен быть централизованным и единым для всех сущностей.  2. **Технологии и стек:**2.1 **Бэкенд:**  * **Язык программирования:**   + **Python/Go/Node.js** — это одни из наиболее популярных языков для создания высокоэффективных веб-сервисов. Python хорошо подходит для быстрого прототипирования и разработки, но Go или Node.js будут предпочтительнее для обработки больших объемов запросов из-за их производительности в асинхронной и многозадачной среде. * **Фреймворк:**   + **FastAPI (Python)** или **Express (Node.js)** — оба фреймворка позволяют создать RESTful API с хорошей производительностью, поддерживают асинхронную обработку запросов и масштабируемость. * **База данных:**   + **PostgreSQL** — основная база данных для хранения информации о лайках, пользователях и сущностях.   + **Redis** — для кэширования частых запросов и для хранения промежуточных данных, например, когда нужно хранить количество лайков на очень активных страницах (кэширование и инкремент в реальном времени).   + **Kafka/ RabbitMQ** — для асинхронной обработки данных, если лайки/дизлайки обрабатываются с задержкой (например, для аналитики или обновления статистики).  2.2 **Фронтенд:**  * **JavaScript/TypeScript (React/Vue.js)** — для взаимодействия с пользователем на клиентской стороне. Модуль лайков может быть частью компонентов на странице, с асинхронными запросами к серверу (например, через fetch или axios). * **WebSockets (если необходимо):** Для обновления состояния лайков в реальном времени без перезагрузки страницы.  2.3 **Интеграция и API:**  * **REST API** — для реализации серверных методов: POST /like, POST /dislike, GET /likes\_count. * **Webhooks** — если нужно уведомлять другие системы о событиях (например, когда пользователь поставил лайк, другая система может обновить статистику). * **OAuth или JWT (JSON Web Tokens)** для аутентификации и авторизации пользователей, если необходимо.  3. **Архитектура системы:**  * **Модульность**: Сервис лайков должен быть легко интегрируемым в другие модули проекта. Логика лайков и дизлайков для каждой сущности будет централизованной, но иметь разные таблицы/модели для разных сущностей. * **Микросервисы:** Если проект имеет микросервисную архитектуру, сервис лайков может быть отдельным микросервисом, который взаимодействует с другими сервисами через API или через очередь сообщений (например, Kafka).   Пример структуры:   * **/likes-service**   + /api/v1/likes/     - POST /{entity\_type}/{entity\_id}/like/ — поставить лайк для сущности.     - POST /{entity\_type}/{entity\_id}/dislike/ — поставить дизлайк.     - GET /{entity\_type}/{entity\_id}/likes/ — получить количество лайков/дизлайков.   + /models     - Like — модель для лайков, которая будет содержать: user\_id, entity\_id, entity\_type, timestamp.   + /utils     - Кэширование, очереди, обработка ошибок.  4. **Хранение данных:**4.1 **База данных**  * **Таблица** likes**:**   + id (PK)   + user\_id (FK, идентификатор пользователя)   + entity\_id (Идентификатор сущности, например, статьи или комментария)   + entity\_type (тип сущности, например, "post", "comment")   + like\_status (boolean или tinyint: 1 — лайк, -1 — дизлайк, 0 — нейтрально)   + timestamp (время действия) * **Таблица** like\_counts**:**   + entity\_id   + entity\_type   + likes\_count (количество лайков)   + dislikes\_count (количество дизлайков)   Для каждого типа сущности (посты, комментарии) таблица like\_counts будет хранить агрегированные данные. Это поможет снизить нагрузку на базу данных, ведь не нужно каждый раз перебирать все лайки для каждой сущности. 4.2 **Кэширование с Redis:** Использование Redis для хранения промежуточных данных о лайках в реальном времени (например, если количество лайков меняется часто), чтобы быстро отдавать эти данные пользователю, не запрашивая базу данных. 5. **Обработка нагрузки и производительность:**  * **Очереди сообщений (Kafka/RabbitMQ)**: Все события лайков и дизлайков могут быть отправлены в очередь для асинхронной обработки, например, для аналитики или дальнейших вычислений. * **Скалируемость:** Для масштабирования можно использовать контейнеризацию (например, Docker и Kubernetes), чтобы разворачивать сервисы на нескольких инстансах и обрабатывать большое количество запросов.   **Пример обработки запроса:**   1. Пользователь кликает на кнопку "Лайк" на странице. 2. Клиентский фронтенд отправляет запрос к API. 3. Бэкенд проверяет, был ли уже поставлен лайк этим пользователем для данной сущности. 4. Если нет, то:    * Обновляется таблица лайков.    * Увеличивается счетчик лайков для сущности.    * Обновляется кеш в Redis. 5. В случае большого трафика или для выполнения дополнительной логики (например, уведомлений или аналитики), событие отправляется в очередь сообщений для дальнейшей обработки.  6. **Мониторинг и логирование:**  * Использование инструментов мониторинга, таких как **Prometheus** и **Grafana**, для отслеживания производительности сервиса, количества запросов и времени отклика. * **ELK Stack (Elasticsearch, Logstash, Kibana)** для логирования и анализа ошибок.  Заключение: Такой подход позволит создать гибкую, масштабируемую и легко интегрируемую систему лайков и дизлайков, способную обрабатывать миллион и более экшнов в день. Применение кэширования, очередей сообщений и баз данных, а также организация микросервисной архитектуры обеспечит необходимую производительность и масштабируемость системы. |

1. Вопрос по логике

У вас есть хранимые изображения на жестком диске. их количество более 1млн, при этом вес каждой от 10 до 50 мб. Изображения постоянно дополняются. До этого момента они не были никак каталогизированны. В настоящий момент надо организовать, при каждом новом пополнении выполняется определение похожих картинок и пометка данной категории.

Как вы технически будете это организовывать?

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Чтобы реализовать систему, которая будет обрабатывать, каталогизировать и определять схожесть изображений при их пополнении, нам нужно организовать несколько ключевых шагов, начиная от хранения изображений до извлечения признаков и поиска похожих изображений. Основные шаги:  1. **Хранение изображений**: Мы будем хранить изображения на жестком диске и поддерживать метаданные в базе данных. 2. **Извлечение признаков изображений**: Для поиска похожих изображений будем использовать нейронные сети и извлекать из них признаки. 3. **Поиск похожих изображений**: Применим алгоритм поиска ближайших соседей для сравнения изображений по их признакам. 4. **Категоризация изображений**: Для определения категории изображения будем использовать классификатор, который будет основан на признаках изображений. 5. **Обновление базы данных и индекса**: Когда новое изображение добавляется, обновляем метаданные и индексируем признаки.  Технологии:  * **Node.js** для серверной части. * **TensorFlow.js** или **ONNX.js** для работы с нейронными сетями на сервере (для извлечения признаков изображений). * **PostgreSQL** или **MongoDB** для хранения метаданных изображений. * **FAISS** (если будем использовать поисковые системы) или **Annoy** для поиска схожих изображений.  Решение с использованием **Node.js**:1. Установка зависимостей Для работы с изображениями, мы можем использовать библиотеку **sharp** (для обработки изображений) и **@tensorflow/tfjs-node** (для нейронных сетей).   |  | | --- | | bash | | npm install sharp @tensorflow/tfjs-node express multer pg |  * **sharp — для обработки изображений (например, изменение размера).** * **@tensorflow/tfjs-node** — для работы с предобученными моделями TensorFlow (например, извлечение признаков из изображения). * **express** — для создания API. * **multer** — для загрузки изображений на сервер. * **pg** — для работы с PostgreSQL.   2. Сервер на **Express.js**  Создадим сервер на Express для загрузки изображений и их обработки.   |  | | --- | | javascript | | const express = require('express');  const multer = require('multer');  const path = require('path');  const sharp = require('sharp');  const tf = require('@tensorflow/tfjs-node');  const { Pool } = require('pg');  // Инициализация базы данных PostgreSQL  const pool = new Pool({  user: 'postgres',  host: 'localhost',  database: 'image\_catalog',  password: 'password',  port: 5432,  });  // Инициализация Express  const app = express();  const port = 3000;  // Мултер для загрузки файлов  const storage = multer.diskStorage({  destination: (req, file, cb) => {  cb(null, './uploads'); // Папка для хранения изображений  },  filename: (req, file, cb) => {  cb(null, Date.now() + path.extname(file.originalname)); // Уникальное имя файла  },  });  const upload = multer({ storage });  // Загрузка модели для извлечения признаков  let model;  async function loadModel() {  model = await tf.loadGraphModel('file://./model/model.json'); // Замените на путь к модели  }  loadModel();  // Извлечение признаков из изображения с использованием модели  async function extractFeatures(imagePath) {  const imageBuffer = await sharp(imagePath).resize(224, 224).toBuffer();  const imageTensor = tf.node.decodeImage(imageBuffer);  const resizedImage = imageTensor.expandDims(0).toFloat().div(tf.scalar(255));    const features = model.predict(resizedImage);  return features.dataSync(); // Возвращаем вектор признаков  }  // API для загрузки и обработки изображений  app.post('/upload', upload.single('image'), async (req, res) => {  try {  const imagePath = req.file.path; // Путь к загруженному изображению  // Извлекаем признаки  const features = await extractFeatures(imagePath);  // Сохраняем изображение и его признаки в базе данных  const client = await pool.connect();  const insertQuery = 'INSERT INTO images(image\_path, features) VALUES($1, $2)';  await client.query(insertQuery, [imagePath, JSON.stringify(features)]);  client.release();  res.status(200).send('Изображение успешно загружено и обработано!');  } catch (err) {  console.error(err);  res.status(500).send('Ошибка при загрузке изображения.');  }  });  // Запуск сервера  app.listen(port, () => {  console.log(`Сервер запущен на порту ${port}`);  }); |  3. Хранение признаков и метаданных в базе данных Мы сохраняем путь к изображению и его вектор признаков в базе данных PostgreSQL. Для этого создаём таблицу:   |  | | --- | | sql | | CREATE TABLE images (  image\_id SERIAL PRIMARY KEY,  image\_path VARCHAR(255) NOT NULL,  features JSONB, -- Вектор признаков изображения в формате JSON  uploaded\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  ); |   Вектор признаков будет храниться в поле features в формате JSON. Каждый раз при загрузке нового изображения в базу данных будут добавляться его признаки.  4. Поиск похожих изображений  Чтобы находить похожие изображения, будем использовать **косинусное расстояние** для сравнения векторов признаков. Создадим API для поиска изображений, похожих на загруженное.   |  | | --- | | javascript | | // Функция для вычисления косинусного расстояния между двумя векторами  function cosineSimilarity(a, b) {  const dotProduct = a.reduce((sum, val, index) => sum + val \* b[index], 0);  const normA = Math.sqrt(a.reduce((sum, val) => sum + val \* val, 0));  const normB = Math.sqrt(b.reduce((sum, val) => sum + val \* val, 0));  return dotProduct / (normA \* normB);  }  // API для поиска похожих изображений  app.post('/search', upload.single('image'), async (req, res) => {  try {  const imagePath = req.file.path;    // Извлекаем признаки из загруженного изображения  const features = await extractFeatures(imagePath);  // Получаем все изображения из базы данных  const client = await pool.connect();  const result = await client.query('SELECT image\_id, features FROM images');  client.release();  // Ищем похожие изображения  const similarities = result.rows.map((row) => {  const dbFeatures = JSON.parse(row.features);  const similarity = cosineSimilarity(features, dbFeatures);  return { image\_id: row.image\_id, similarity };  });  // Сортируем по схожести  similarities.sort((a, b) => b.similarity - a.similarity);  // Отправляем наиболее похожие изображения  res.json(similarities.slice(0, 5)); // Отправляем топ 5 похожих изображений  } catch (err) {  console.error(err);  res.status(500).send('Ошибка при поиске похожих изображений.');  }  }); |   5. Вывод  Теперь у нас есть несколько ключевых функций:   1. **Загрузка изображений** с использованием **Multer**. 2. **Извлечение признаков** с использованием предобученной модели. 3. **Сохранение изображений и их признаков** в базу данных. 4. **Поиск похожих изображений** с использованием косинусного расстояния между векторами признаков.  Масштабирование Для обработки миллионов изображений можно:   * Использовать **облачное хранилище** (например, AWS S3) для хранения изображений. * Использовать **индексацию** в базе данных или специализированных системах поиска (например, **FAISS** или **Annoy**) для более быстрого поиска по признакам.  Заключение Это решение позволяет эффективно обрабатывать большие объёмы изображений, извлекая признаки, классифицируя их и находя схожие изображения. |