tech ПРОЕКТ МЭРА МОСКВЫ ДЕПАРТАМЕНТ ПРЕДПРИНИМАТЕЛЬСТВА И ИННОВАЦИОННОГО РАЗВИТИЯ ГОРОДА МОСКВЫ РАЗВИТИЕ ЧЕЛОВЕЧЕСКОГО КАПИТАЛА ЛИДЕРЫ ЦИФРОВОЙ ТРАНСФОРМАЦИИ

Система рекомендаций по оптимизации производительности Data Lakehouse

КОМАНДА “MLexus”

О команде

- г. Москва
- 5 человек в команде
- Капитан команды: Кудрявцев Ярослав

Наименование задачи:

Система рекомендаций по оптимизации производительности Data Lakehouse

Описание решения:

Интеллектуальный ассистент для оптимизации Data Lakehouse, анализирующий метаданные и статистику запросов через LLM для генерации actionable-рекомендаций: рефакторинг SQL, оптимизация физической структуры (партиционирование/кластеризация), упрощение запросов, подбор легковесной LLM для экономии ресурсов, защита запросов от утечек и инъекций, кластеризация схожих запросов для групповой оптимизации и рассмотрены варианты внутреннего бенчмарка.

Как планируется дальше использовать и развивать решение

1. Превратить решение в ведущую SaaS-платформу для автоматической оптимизации Data Lakehouse.
2. Развитие, направленное на расширение рекомендаций по управлению ресурсами и оптимизации затрат, а также на создание самообучающейся системы, постоянно улучшающейся за счет обратной связи.
3. Рассматриваем запуск отраслевого бенчмарка для сообщества.

КОМАНДА “MLexus”

Егор Шипилов

- ML-engineer
- @Eg0Mak
- +7 967 234 1334

Никита Кирнасов

- Fullstack-разработчик
- @Dcomb21
- +7 902 213 4506

Владислав Апухтин

- Data Scientist
- @kyl0q
- +7 915 062 2993

Ярослав Кудрявцев

- Data Analyst
- @DFAYM
- +7 915 255 6747

Фёдор Дубовицкий

- Data Storyteller
- @chupikbongo
- +7 925 319 1842

КОМАНДА “MLexus”

Краткая история команды:

Наша команда сформировалась в стенах Финансового университета при Правительстве РФ, где мы являемся студентами одной учебной группы. Это мероприятие стало для нас первым опытом участия в хакатоне. Мы нацелены на продуктивную работу, применение полученных знаний на практике и достижение конкурентоспособного результата.

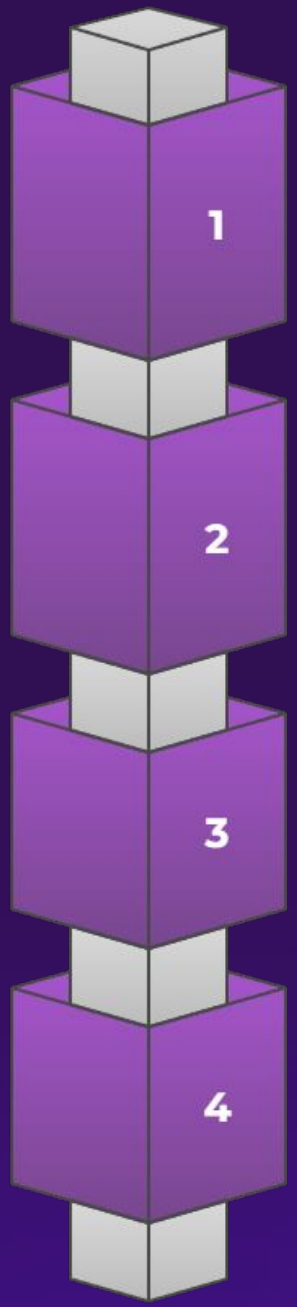
Почему вы выбрали именно эту задачу из предложенных на хакатоне?

Мы выбрали эту задачу, так как она полностью соответствует нашим интересам в Data Science и Big Data. Мы стремимся создать практичное решение для оптимизации данных, а также ценим возможность сотрудничества с технологическим лидером VK Tech.

С какими основными сложностями или вызовами вы столкнулись и как их преодолели?

Основными сложностями стали подготовка качественного датасета для обучения и настройка гиперпараметров модели в условиях конфликтов версий библиотек. Мы успешно преодолели это с помощью фреймворка Optuna, который автоматизировал поиск оптимальных параметров для LoRA-адаптера и SFT-трейнера.

ПЛАН РАБОТЫ



Собрали датасеты

Сбор и организация необходимых данных для обучения модели.

Разметили датасеты

Разметка данных для обеспечения точности и последовательности.

Обучили LLM

Обучение модели на подготовленных данных.

Создали API

Разработка интерфейса для доступа к модели.

Внедрили LLM

Интеграция модели LLM в систему для улучшения возможностей обработки языка.

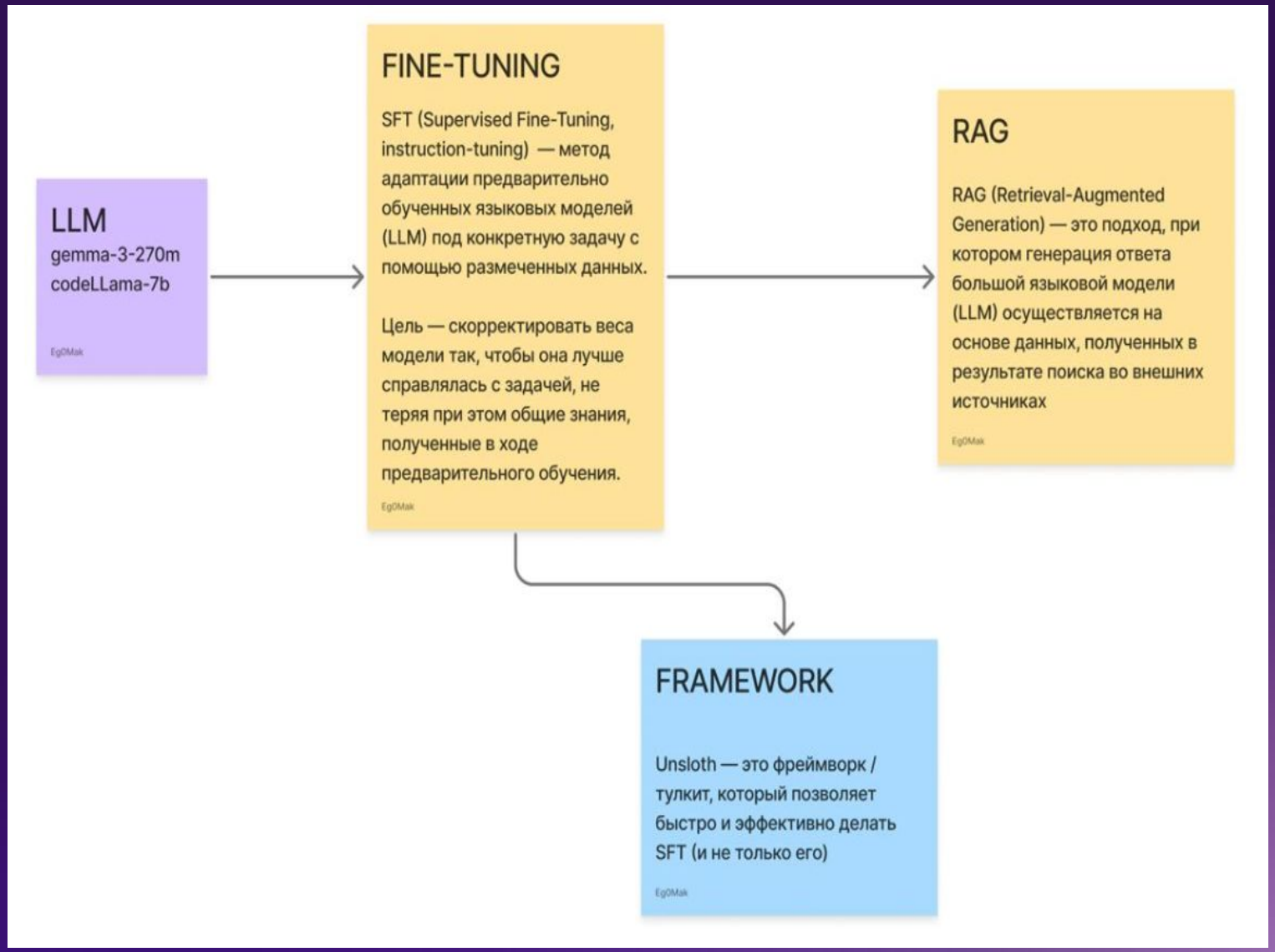
Добавили метрики

Включение метрик для измерения и оценки производительности системы.

Получили итоговый продукт

Достижение завершенного и функционального продукта, готового к использованию.

О МОДЕЛИ



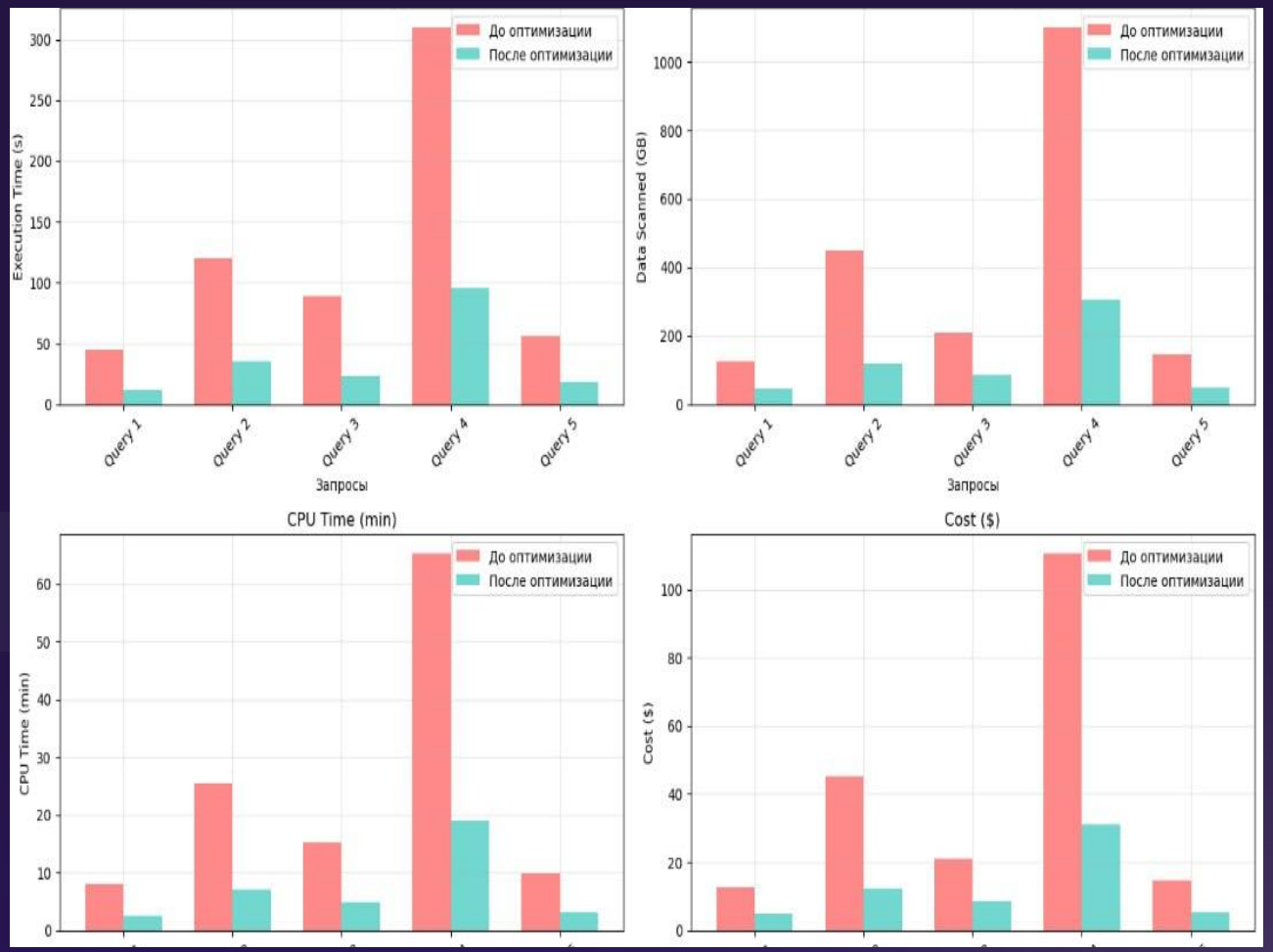
RAG
Retrieval Augmented Generation

LoRA

SFT

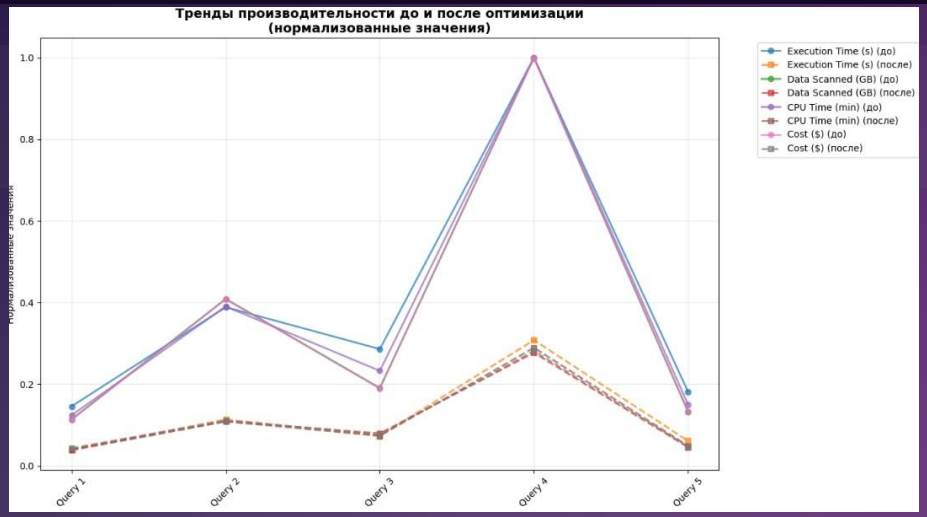
МЕТРИКИ ОПТИМИЗАЦИИ

www.lide



Мы будем вычислять метрики исходного запроса и оптимизированного запроса и выводить их в интерфейсе, чтобы была наглядная разница между ними.

www.lider.co



РЕСТРУКТУРИЗАЦИЯ БАЗЫ ДАННЫХ

Идея бенчмарка и реструктуризации базы данных такая: К нам поступают запросы, их много и все они разные, нам надо оценить насколько они производительны, куда чаще всего обращаются пользователи, сюда относятся вопросы партиционрования таблиц, выделения временных таблиц и перестройка структуры базы данных.

Основная цель, быстро принимать и отдавать данные, при этом надо соблюсти минимальное представление по нагруженности линий передачи данных и сформировать быстрый и легкий иструмент.

Предлагается получать SQL запросы и оценивать их по нескольким параметрам (см. ниже), далее, так как запросы приходят у нас по времени можно рассматривать их как временной ряд (многомерный). По оси X у вас будут временные метки, по оси Y параметры вашего запроса (их можно получить из SQL Trino, он оценивает работу SQL запроса по 150 параметрам). То есть осей Y у вас будет много.

Далее, переводим SQL запросы с временными метками и параметрами во временной ряд и строим по этом ряду граф. Затем оценивая этот граф (метрики по нему смотрит ниже) ищем "ботлнеки" и далее определяем на каком запросе что у нас тормозится и перестраиваем структуру нашей базы данных.

Ниже приведен код по получению временного ряда и построению по нему графа. Таким образом, оценивая такой граф мы в динамическом режиме видим как работает наша база данных по запросам.

При этом можно даже не собирать логи, так как все происходит в режиме реалтайма (можно также собирать для статистики). Вершины графа это SQL запросы с параметрами, ребра это связи между запросами. По SQL запросам можно восстановить структуру базы данных.

В тестовом варианте мы представили временные промежутки длинные, но их можно изменить на любые вплоть до 1 сек или 1 мкр. сек. Поддерживается самый широкий диапазон тайминга. Если два или более запроса приходят одновременно ставится метка, также можно просматривать логи от отдельного пользователя, то есть его запросы.

ПАРАМЕТРЫ SQL ЗАПРОСОВ

- Время выполнения (Execution Time): Измеряется время, затраченное на полное выполнение запроса, включая этапы планирования, обработки и вывода результата.
- План запроса (Query Plan): Структура плана исполнения запроса помогает понять, как база данных решает выполнить запрос (использует ли индекс, какой тип соединения применяется и т.д.).
- Используемые индексы (Indexes Used): Проверка, используются ли индексы для ускорения операций чтения и фильтрации.
- Объем обработанных данных (Rows Scanned / Rows Examined): Сколько строк было фактически проверено базой данных перед выдачей результата.
- Эффективность сортировки (Sort Operations): Оценка количества операций сортировки и их влияние на общую производительность.
- Количества соединений (Joins): Анализ того, сколько таблиц участвует в соединениях и насколько эффективны используемые методы соединения.
- Загрузка ресурсов сервера (CPU, Memory Usage): Мониторинг нагрузки на процессор и память, вызванной выполнением запроса.
- Операции ввода-вывода (Disk I/O): Определение объема дисковых операций, необходимых для выполнения запроса.
- Занимаемая транзакционная нагрузка (Locks Acquired): Общее количество заблокированных объектов (таблиц, строк) и продолжительность блокировки.
- Параметр использования памяти (Memory Consumption): Объем оперативной памяти, потребляемый процессом выполнения запроса.
- Тип операции (Operation Type): Является ли запрос простым выбором данных (SELECT), изменением данных (INSERT, UPDATE, DELETE) или сложной комбинацией операций.
- Количество используемых сессий (Sessions Involved): Если запрос запускается параллельно несколькими пользователями или сеансами, можно измерить нагрузку на систему от параллельных подключений.
- Процент использования кеширования (Cache Hit Ratio): Насколько сильно используется кэшированный результат предыдущего выполнения запроса.
- и так далее...

ТЕКСТОВЫЙ ПРИМЕР: КОЛЧЕСТВО СТРОК В ЗАПРОСЕ

Каждый тикер представляет собой таблицу в базе данных, на каждую таблицу приходят запросы с параметрами. Мы оцениваем эти пареметры, например, колличество строк в запросе (можно любые другие параметры которые указаны выше в таблице №1) и ставим метку, так фомируем временной ряд где по оси X отложено значение времени, по оси Y суммарный показатель параметра.

то есть,

id_запроса: 1

Текст_запроса: SELECT user_id, COUNT(*) AS orders_count 6 FROM orders WHERE order_date >= '2025-01-01' GROUP BY user_id ORDER BY orders_count DESC;

Колличество_строк: 5

id_запроса: 2

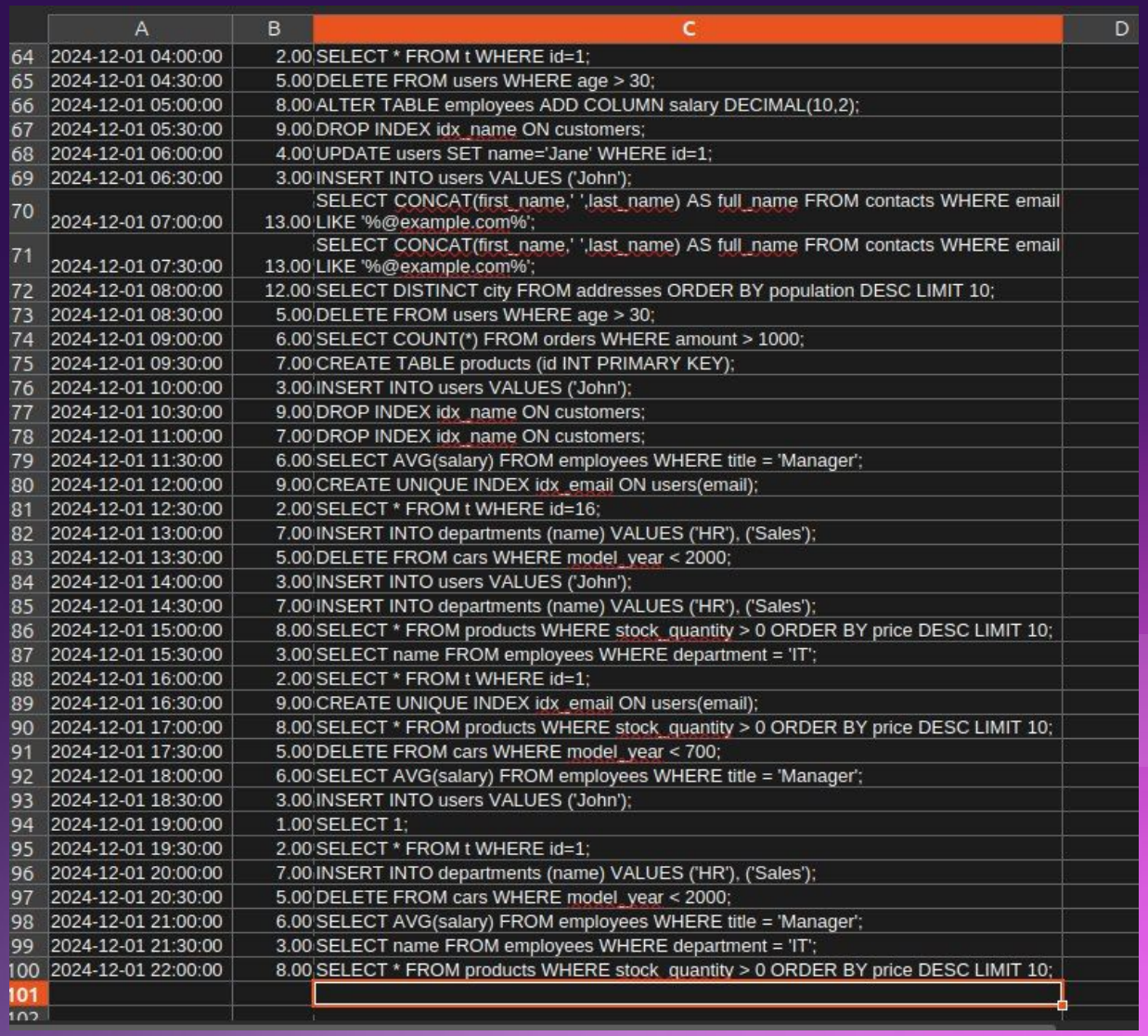
Текст_запроса: INSERT INTO products (product_name, price, category) VALUES ('Smartphone', 699.99, 'Electronics');

Колличество_строк: 2

id_запроса: 3

UPDATE employees SET salary = salary * 1.1 WHERE department = 'Sales' AND hire_date < '2020-01-01';

Колличество_строк: 3



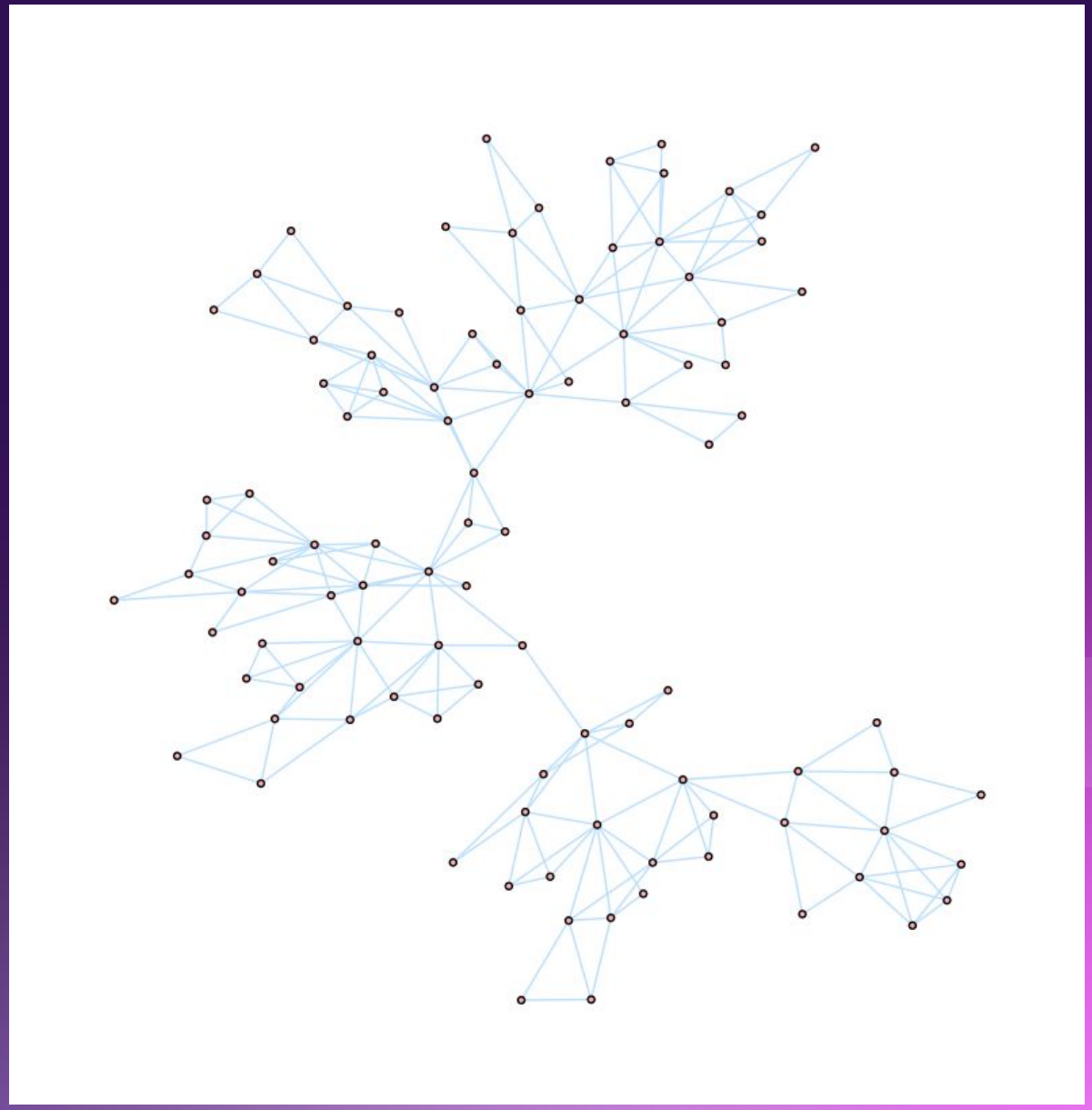
	A	B	C	D
64	2024-12-01 04:00:00	2.00	SELECT * FROM t WHERE id=1;	
65	2024-12-01 04:30:00	5.00	DELETE FROM users WHERE age > 30;	
66	2024-12-01 05:00:00	8.00	ALTER TABLE employees ADD COLUMN salary DECIMAL(10,2);	
67	2024-12-01 05:30:00	9.00	DROP INDEX idx_name ON customers;	
68	2024-12-01 06:00:00	4.00	UPDATE users SET name='Jane' WHERE id=1;	
69	2024-12-01 06:30:00	3.00	INSERT INTO users VALUES ('John');	
70	2024-12-01 07:00:00	13.00	SELECT CONCAT(first_name,' ',last_name) AS full_name FROM contacts WHERE email LIKE '%@example.com%';	
71	2024-12-01 07:30:00	13.00	SELECT CONCAT(first_name,' ',last_name) AS full_name FROM contacts WHERE email LIKE '%@example.com%';	
72	2024-12-01 08:00:00	12.00	SELECT DISTINCT city FROM addresses ORDER BY population DESC LIMIT 10;	
73	2024-12-01 08:30:00	5.00	DELETE FROM users WHERE age > 30;	
74	2024-12-01 09:00:00	6.00	SELECT COUNT(*) FROM orders WHERE amount > 1000;	
75	2024-12-01 09:30:00	7.00	CREATE TABLE products (id INT PRIMARY KEY);	
76	2024-12-01 10:00:00	3.00	INSERT INTO users VALUES ('John');	
77	2024-12-01 10:30:00	9.00	DROP INDEX idx_name ON customers;	
78	2024-12-01 11:00:00	7.00	DROP INDEX idx_name ON customers;	
79	2024-12-01 11:30:00	6.00	SELECT AVG(salary) FROM employees WHERE title = 'Manager';	
80	2024-12-01 12:00:00	9.00	CREATE UNIQUE INDEX idx_email ON users(email);	
81	2024-12-01 12:30:00	2.00	SELECT * FROM t WHERE id=16;	
82	2024-12-01 13:00:00	7.00	INSERT INTO departments (name) VALUES ('HR'), ('Sales');	
83	2024-12-01 13:30:00	5.00	DELETE FROM cars WHERE model_year < 2000;	
84	2024-12-01 14:00:00	3.00	INSERT INTO users VALUES ('John');	
85	2024-12-01 14:30:00	7.00	INSERT INTO departments (name) VALUES ('HR'), ('Sales');	
86	2024-12-01 15:00:00	8.00	SELECT * FROM products WHERE stock_quantity > 0 ORDER BY price DESC LIMIT 10;	
87	2024-12-01 15:30:00	3.00	SELECT name FROM employees WHERE department = 'IT';	
88	2024-12-01 16:00:00	2.00	SELECT * FROM t WHERE id=1;	
89	2024-12-01 16:30:00	9.00	CREATE UNIQUE INDEX idx_email ON users(email);	
90	2024-12-01 17:00:00	8.00	SELECT * FROM products WHERE stock_quantity > 0 ORDER BY price DESC LIMIT 10;	
91	2024-12-01 17:30:00	5.00	DELETE FROM cars WHERE model_year < 700;	
92	2024-12-01 18:00:00	6.00	SELECT AVG(salary) FROM employees WHERE title = 'Manager';	
93	2024-12-01 18:30:00	3.00	INSERT INTO users VALUES ('John');	
94	2024-12-01 19:00:00	1.00	SELECT 1;	
95	2024-12-01 19:30:00	2.00	SELECT * FROM t WHERE id=1;	
96	2024-12-01 20:00:00	7.00	INSERT INTO departments (name) VALUES ('HR'), ('Sales');	
97	2024-12-01 20:30:00	5.00	DELETE FROM cars WHERE model_year < 2000;	
98	2024-12-01 21:00:00	6.00	SELECT AVG(salary) FROM employees WHERE title = 'Manager';	
99	2024-12-01 21:30:00	3.00	SELECT name FROM employees WHERE department = 'IT';	
100	2024-12-01 22:00:00	8.00	SELECT * FROM products WHERE stock_quantity > 0 ORDER BY price DESC LIMIT 10;	
101				

ПОСТРОЕНИЕ ГРАФА БАЗЫ ДАННЫХ ПО ЗАПРОСАМ

Функция visibility_graph строит граф видимости для временного ряда. Она последовательно проверяет каждую пару точек на наличие прямой видимости друг относительно друга, используя геометрическое условие: точка считается видимой, если ни одна промежуточная точка не находится выше линии, соединяющей рассматриваемые точки. Если условие выполняется, добавляется ребро между этими вершинами графа. Таким образом, полученный граф отражает структуру временных зависимостей и взаимосвязей внутри исходного ряда данных.

Каждая точка нашего графа представляет количество строк в отдельных SQL-запросах. Алгоритм построения графа видимости позволяет выявить зависимости и связи между различными запросами по количеству возвращаемых ими строк. Например, если два запроса имеют прямую видимую связь (ребро в графе), это означает, что число строк в одном запросе непосредственно влияет на количество строк другого запроса, либо они оба зависят от общего набора условий выборки. Такой подход помогает анализировать закономерности и паттерны распределения объемов данных, выявлять аномалии или кластеры запросов с похожими характеристиками.

Судя по этому графу у нас есть три явные кластера запросов, оценивая параметры этого графа можно сказать, надо пересобирать базу или нет.



ПАРАМЕТРЫ ГРАФА

 # 1. Минимальное расстояние между вершинами (Minimum distance between vertices, MDBV)
 # 2. Дисперсия длин рёбер (Dispersion of edge lengths, DEL)
 # 3. Максимальное отклонение углов (Maximum angle deviation, MAD)
 # 4. Площадь покрытия (Coverage area, CA)
 # 5. Расстояние центров масс компонентов (Distance of centers of mass of components, DCMC)
 # 6. Симметрия (Symmetry, Sym)
 # 7. Регулярность формы (Regularity of form, RF)
 # 8. Соотношение сторон окна (Window aspect ratio, WR)
 # 9. Среднее абсолютное отклонение длины рёбер (Mean Absolute Edge Length Deviation, MAELD)
 # 10. Огибающая линия контура графа (Graph Contour Envelope, GCE)
 # 11. Угол поворота (Angle Preservation Metric, APM)
 # 12. Критерий корреляции взаиморасположения (Proximity Correlation Criterion, PCC)
 # 13. Показатель осевого выравнивания (Axis Alignment Index, AAI)
 # 14. Относительная разница между максимальным и средним расстоянием (Relative Distance Difference Ratio, RDDR)
 # 15. Размер краевых выпуклых оболочек (Convex Hull Size of Edges, CHSE)
и так далее...

МОДЕЛЬ ПЕРЕУПАКОВКИ ГРАФА И ВОССТАНОВЛЕНИЕ СТРУКТУРЫ БАЗЫ

Имея набор параметров можно оценить где по графу есть проблеммы.

Для поиска (bottlenecks) в нашем графе можно использовать анализ центральности вершин (degree centrality, betweenness centrality, closeness centrality и прочие параметры), выявление плотных подграфов и распределённых путей. Особое внимание уделяем параметрам минимального расстояния между вершинами (MDBV), средней длине рёбер (MAELD) и относительной разницы расстояний (RDDR), так как они помогают выявить участки с максимальной нагрузкой и длительными путями прохождения запросов, определяющими узкие места сети.

Здесь мы пытаемся сделать переупаковку нашего графа и понять как его лучше разместить то есть уложить чтобы пересобрать базу данных, то есть мы пошли от метода восстановления запросов по пути к восстановлению структуры базы по ним. При этом используем графовые нейронные сети.

```
    layout_algorithms = ["kamada_kawai"]
    for g in graph.values():
        for layout_algorithm in layout_algorithms:
            data = create_graph_data(g,layout_algorithm)
            data_list.append(data)

    batch_size = 16
    learning_rate = 0.0000001
    num_epochs = 15
    num_node_features = data_list[0].x.shape[1]

    def objective(trial):
        num_layers = trial.suggest_int('num_layers', 2, 4, 8)
        hidden_dim = trial.suggest_categorical('hidden_dim', [32, 64, 128, 256])
        num_heads = trial.suggest_categorical('num_heads', [4, 8, 16, 32])

        model = GraphLayoutEvaluator(num_node_features=num_node_features,
                                     num_layers=num_layers,
                                     hidden_dim=hidden_dim,
                                     num_heads=num_heads)
        optimizer = torch.optim.Adam(model.parameters(),
                                      lr=learning_rate,
                                      #betas=(beta1, beta2),
                                      #eps=epsilon,
                                      #weight_decay=l2_penalty,
                                      amsgrad=False)
        model.train()
```