Санкт-Петербургский государственный университет

КУТЯВИН Денис Максимович

Выпускная квалификационная работа

Оптимизация хранения официальных документов

Уровень образования: бакалавриат
Направление 02.03.01 Математика и компьютерные науки
Основная образовательная программа СВ.5152.2019
Математика, алгоритмы и анализ данных

Научный руководитель:
Доцент, Факультет математики и компьютерных наук СПбГУ, Авдюшенко Александр Юрьевич
Рецензент:
Инженер-программист, Общество с ограниченной ответственностью «БалтИнфоКом», Козлов Вадим Константинович

Санкт-Петербург

2023

Содержание

Введение

Мотивация

Люди постоянно взаимодействуют между собой, эти взаимодействия надо как-то регламентировать и записывать, именно к этому пришла наша цивилизация. Для этого используются документы. Что такое документ? Это отдельный большой вопрос. Давайте смотреть на него как на некоторый объект, хранящий информацию.

Также он обладает некоторыми важными свойствами, одно из которых это то, что документ нельзя уничтожить (так как он регламентирует какое-то событие или его возможность), а чтобы что-то изменить надо выпустить новый. Так для одного человека может накапливаться огромная куча документов из разных сфер жизни. Чем важна работа с такими документами? Можно привести множество примеров, приведу один самый показательный: если бы все документы хранились бы в одном месте с удобным доступом, то для получения кредита необходимо было бы всего лишь предоставить доступ, а дальше банк быстро мог бы обработать их и выписать разрешение, ещё удобнее было бы это для более сложных сделок где наборы документов нужны в различных местах.

Еще одной важной особенностью документов является их зависимость, отношения между людьми могут быть горизонтальными и вертикальными, так некоторые действия либо согласуются, либо разрешаются, аналогично и для документов, чтобы один документ был действителен нужна ссылка на другой подтверждающий или разрешающий документ, таким образом создаются целые сети документов, которые зависимы друг от друга. Для одного человека это не такое большое количество, а вот для компаний с кучей нормативных актов зависимости могут очень сильно разрастаться.

Таким образом создаётся огромное количество зависимых друг от друга данных определенного вида.

В разных сферах жизни документы хранятся в разных местах, поэтому давайте посмотрим на документацию и зависимости одной фирмы. Раньше

документы хранились в бумажном виде и со временем отправлялись со временем в архив, сейчас же по мере цифровизации всё больше документов хранятся в электронном виде в различных базах данных, чаще всего это простые реляционные базы данных, да там есть зависимости, но оптимальность данного подхода оставляет желать лучшего, так как для вычисления даже самых простых запросов необходимо будет пробегать по всей базе данных и иногда даже не один раз, так как наши зависимости могут аннулировать ранее существующие документы и это надо учитывать. Конечно, иногда с этим можно побороться раздуванием таблицы и учётом этих самых зависимостей, как отдельных параметров, но эффективность и размеры сильно страдают при увеличении количества документов, поэтому сейчас мы рассмотрим несколько другой подход к данной проблеме хранения официальных документов и их обработке с целью получения данных.

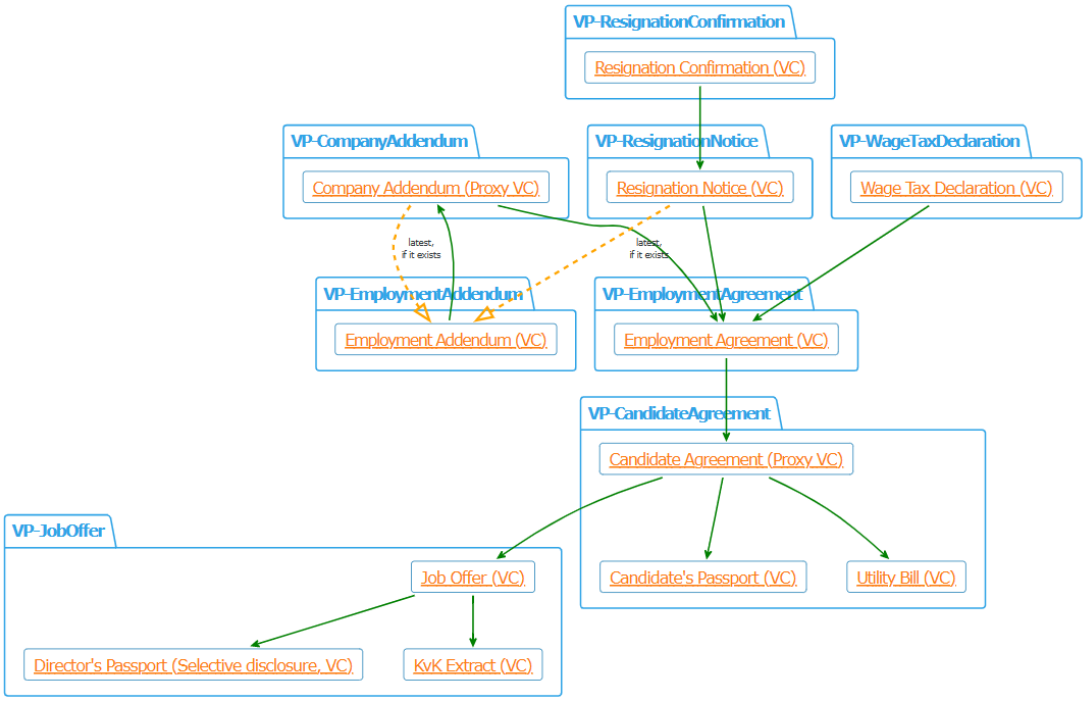
Технические особенности

Посмотрим на это всё, как на граф, в нём вершины это документы, хранящие некоторую информацию, а ориентированные ребра это уже ссылки между ними. Далее для построения нашей модели надо понять, как удобно хранить документы, так как они относятся к Linked Data, то есть к структурированным данным, которые взаимосвязаны с другими данными. Многие данные сейчас передаются в формате JSON (текстовый формат обмена данными, основанный на JavaScript), но нам нужна немного более усовершенствованная версия, созданная специально для взаимосвязанных данных, а именно JSON-LD. Она удобна тем, что изначально Одной из целей JSON-LD было потребовать от разработчиков как можно меньше усилий для преобразования существующего JSON в JSON-LD.

Но это всё же официальные документы и они должны иметь под собой некоторый гарант подлинности. Для этого в документах используется Verifiable Credentials Data Model, с её помощью документ содержит специальную цифровую подпись подтверждающую защищенные данные.

Таким образом мы будем иметь модель документа, который можно будет проверить на подлинность. а так же в нём будет содержаться ссылка на другой документ.

Осталось разобраться со структурой нашего графа, то есть с разрешенными связями. Некоторые ссылки разрешены, а некоторые нет, то есть у нас задаётся некоторая графовая грамматика, по которой можно проверять правильность построения графа документов. Если же собрать все правила вместе, то получим граф разрешённых зависимостей. Ниже приведен граф разрешённых зависимостей с которыми мы будем работать и проводить тесты. Это граф для компании Truvity, содержащий информацию о сотрудниках, то есть документы позволяющие устроить их на работу, документы содержащие информацию об их текущей работе и об изменениях в статусе этой работы (изменение должности, изменение зарплаты, увольнение).



Заметим, что все документы здесь вписываются в систему Verifiable Credentials Data Model v1.1[1].

Понятно, что система документов может быть проще или в десятки раз сложнее с различными зависимостями, но граф по такой грамматике действительно можно будет построить, это большая тема для обсуждения освященная в работе Matrix Graph Grammars, Pedro Pablo Perez Velasco [4].

Реализация проекта

База данных и запросы

Обработка документов

Мы знаем, что документы не удаляются, а значит необходимо добавить в в базу данных все существующие на данный момент документы, а также создать функцию добавления новых документов.

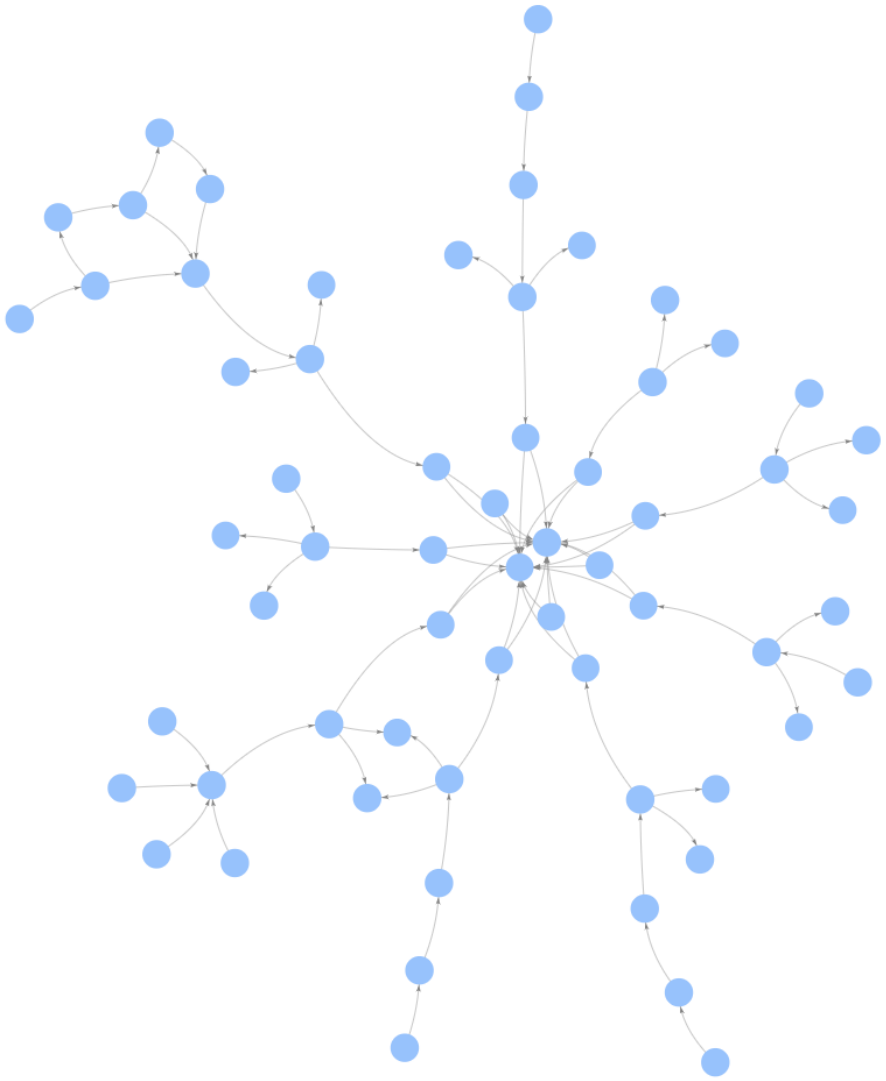
На самом деле мы имеем очень удобную структуру, где для каждой вершины мы имеем данные для нее, т.е. знаем что это за документ, его id и другую необходимую информацию. По этим же id и делаются ссылки между документами. Поэтому алгоритм добавления вершины выглядит так:

1) при обработке стоит проводить верификацию документа
2) вычислять id, тип документа и все ссылки на другие документы
3) создается новая вершина для данного документа
4) проводятся рёбра от новой вершины к вершинам с нужным id, при этом проверяется типовое соответствие ссылок

При расхождении на каком-нибудь этапе необходимо выдать ошибку и удалить данную вершину со всеми ребрами из графа.

Так мы научились добавлять документы, для создания же базы по уже существующему архиву достаточно только отсортировать все документы в порядке создания и тогда можно будет применить модель поочередного создания, так как документы могут ссылаться на только уже существующие.

Ниже можно увидеть граф для компании Truvity, он создан для 8 сотрудников и содержит 72 документов.



Этот граф был создан в Amazon Neptune, так как это удобная среда и язык используемый для запросов это Gremlin.

Запросы

Граф создан, теперь надо понять как построить запрос к данному графу. Разберем это на примере построения запроса, который должен выдать всех активных сотрудников фирмы.

Чтобы сделать такую проверку надо посмотреть на все вершины имеющие type EmploymentAgreement, дальше спустится от них к CandidateAgreemen, а от него к Passport, оттуда мы берём атрибут Surname, для EmploymentAgreement выше проверяем наличие ResignationNotice и RecignationConfirmation, после поочерёдно проверяем JobOffer для CandidateAgrement, сортированные EmploymentAgreement по времени и берём последний, дальше CompanyAddendum и в конце сортированные также по времени EmploymentAddendum, проверяя берём оттуда Position, далее добавляем уже найденную ниже Surname к позиции и выводим.

Другие запросы делаются аналогично исходя из логики запроса.

Тестирование

Посмотрим сколько времени занимает наш запрос и сравним его со временем такого же запроса для наивной реализации, где все документы хранятся в соответствующих таблицах по типам и хранят информацию о ссылках. Также стоит сравнить время создания и баз данных. Понятно, что время зависит и от загруженности системы, так что проведём несколько измерений (в нашем случае 100) для одинаковых запросов и возьмем медианное время.

Измеряемая величина	Наивная реализация	Графовая БД
Время создания	3,41 s	3,99 s
Время на обработку запроса	6,42 s	138 ms

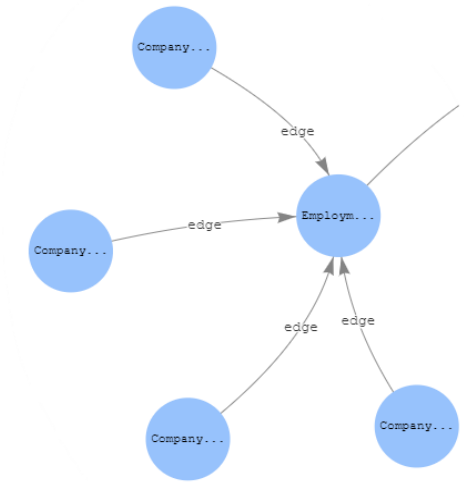
Как можно заметить наивная реализация тратит меньше времени на создание, зато на обработку запроса требуется время уменьшилось на 97%, что важно при росте количества данных.

Тестирование на растущих деревьях

Мы посмотрели на достаточно маленький пример, но что будет, если наша таблица очень сильно разрастётся.

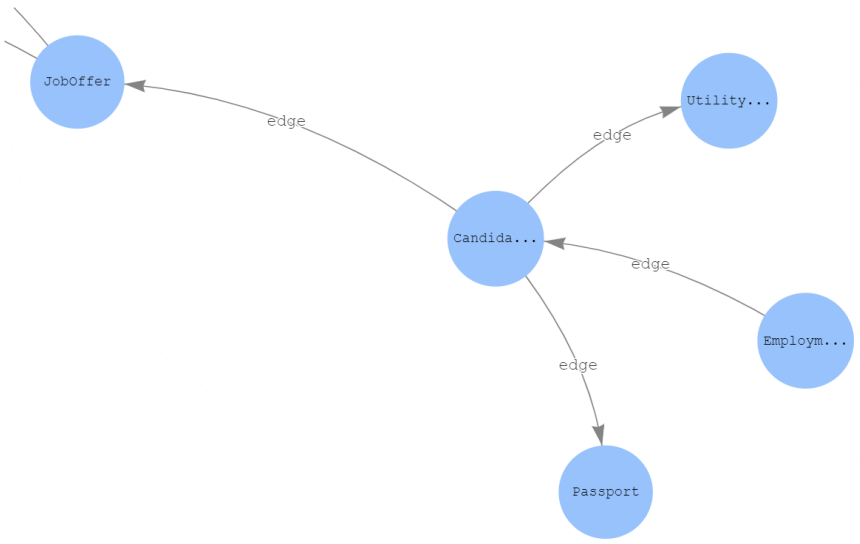
Для этого будем добавлять документы определенного вида:

1) сначала посмотрим, что будет со временем запроса, если мы будем добавлять один или несколько CompanyAddendum к EmploymentAgreement:

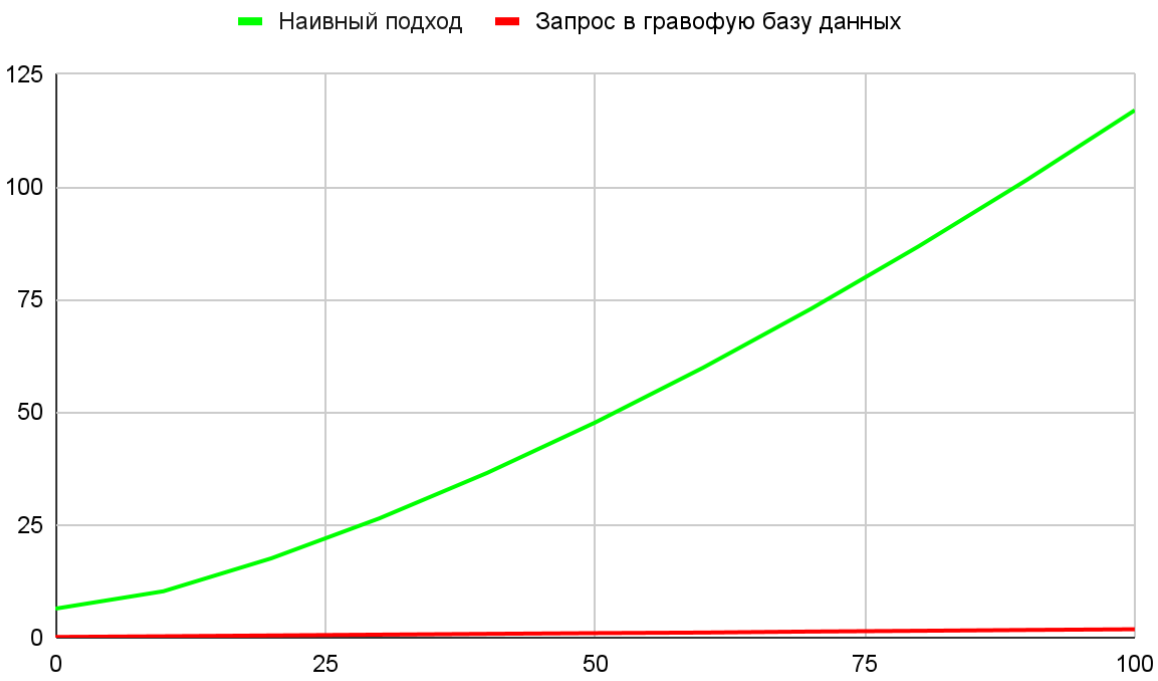


Такими действиями мы изменяем соглашение о трудоустройстве, в нашем случае будем менять должность. Они будут добавляться рандомно к уже существующим EmploymentAgreement. Таких CompanyAddendum будет 1000 штук. После проверки запроса для наивной реализации мы получили среднее значение в 54,3 s, а реализация запроса через графовую базу данных дала среднее значение в 0,145 ms.

2) будем добавлять нового сотрудника, то есть все его атрибуты: JobOffer, CandidateAgrement, EmploymentAgreement, Passport, UtilityBill:



Добавим 100 новых сотрудников таким образом. Запрос будет производиться каждые 10 новых сотрудников.



Как видим для наивной реализации числа очень быстро растут, для запросов в графовую базу данных таких больших чисел не наблюдается, но значение выросло до 1,863 s, что всё равно быстрее чем наивная реализация для 8 сотрудников.

Ускорение работы

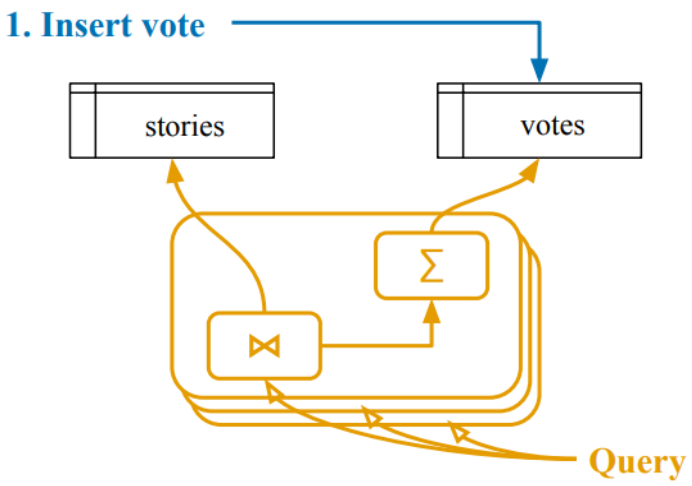
Проведя тестирование было видно, что при разрастании количества сотрудников, а значит и при увеличении документации и сложности её зависимостей общее время на выполнение запроса будет расти и достаточно весомо при больших объемах данным.

Поэтому для борьбы с таким увеличением можно использовать разные методы ускорения работы.

Посмотрим на эти методы и разберем их плюсы и минусы на более простом примере: пусть у нас есть простые записи (stories) и есть голоса за них (votes), мы хотим подсчитать общее число голосов. Это будет аналогом того, что мы например ищем количество сотрудников когда-то работавших в фирме.

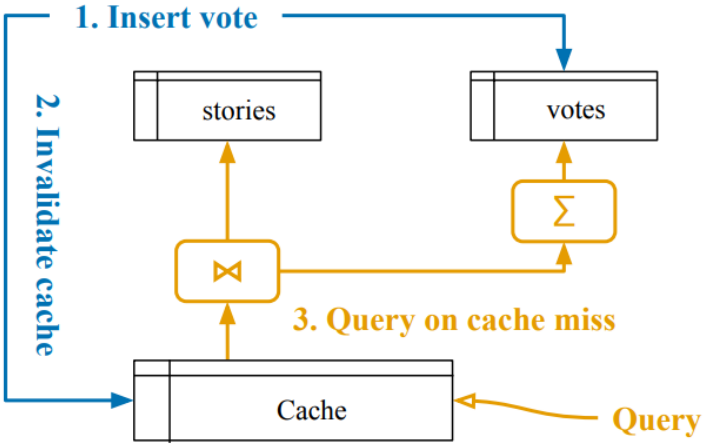
Ниже показано, как запросы приложения функционируют на высоком уровне в традиционной модели: каждый запрос, который выдает приложение, выполняет план запроса,

представленный агрегацией и объединением на рисунке. Несколько одновременных запросы выполняются независимо, даже если они выполняют один и тот же запрос.

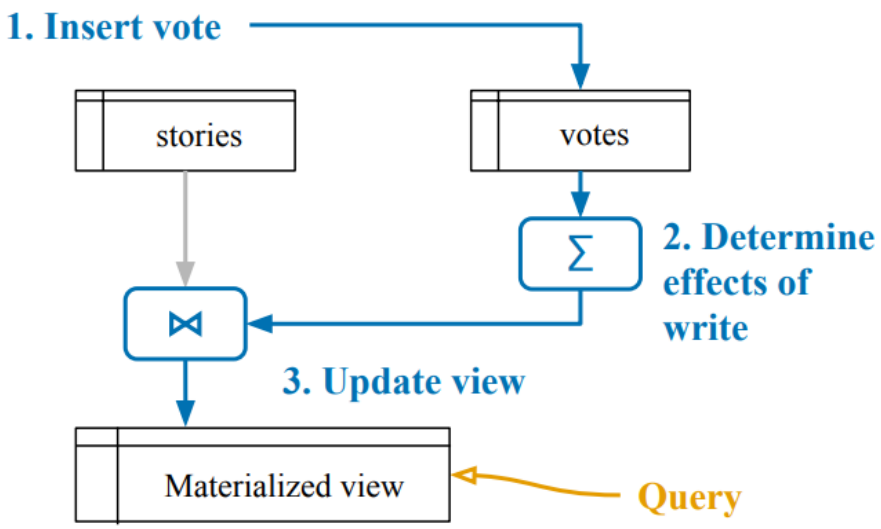


Одним из решений для ускорения является кеш, мы заранее запоминаем ответ на сделанный запрос, если это потребуется, так мы сразу сможем давать ответ на запрос, если не было изменений, а если были, то придется

делать полный запрос к базе данных. Ниже представлена архитектурам работы кеша:



Ещё неплохим подспорьем для кеша может послужить материализованное представление. Система обновляет материализованные результаты в ответ на запись приложения, а при чтении получает доступ только к сохраненным результатам. Ниже представлена архитектура материализованного представления:



Заметим, что использование материализованного представления конечно всегда почти мгновенно отвечает на запрос, но его содержание тратит очень много ресурсов и при увеличении объемов данных становится очень затратным. Кэш же неплохо справляется со своими обязанностями, если не надо считать не сложные метрики проверяемые добавлением

определённых типов файлов, в противном случае нам постоянно придется пересчитывать значение в кэше, что ведет к дополнительным накладкам.
Есть ещё одно решение, это некоторое совмещение плюсов кэширования и материализованного представления, это частичное материализованное представление основанное на потоке данных. Материализованные представления представляют собой “почти готовое” решение для автоматического кэширования. Они обеспечивают отличный базовый механизм для эффективного хранения и сопровождения результатов запросов способом, который хорошо согласуется с тем, как уже работают приложения: путем выдачи запросов. Чего не хватает для того, чтобы материализованные представления стали жизнеспособной заменой специальным стратегиям кэширования, используемым современными приложениями, так это способа сделать материализованные представления более динамичными. В частности, чтобы служить хорошей заменой кэша, материализованные представления должны поддержка эффективного добавления новых запросов и удаления старых результатов во время выполнения. Чтобы восполнить пробел, в этом тезисе предлагается частично материализованное представление, или сокращенно частичное представление. Частичное представление позволяет помечать записи в материализованных представлениях как отсутствующие и вводит дополнительные запросы для вычисления такого отсутствующего состояния по требованию. Это позволяет эффективно добавлять новые запросы, оставляя исходное материализованное представление пустым и заполняя его только в ответ на запросы приложения. Кроме того, по мере того, как приложение теряет интерес к старым результаты запроса, эти результаты могут быть удалены для освобождения памяти, которая, в свою очередь, может быть использована для кэширования более важных результатов запроса. По сути, частичное состояние позволяет материализованным представлениям функционировать подобно кэшам. Модель системы по-прежнему выглядит так, как показано на рисунке для

материализованных представлений выше, за исключением того, что материализованное представление также содержит параметры, значение которых приложение предоставляет во время выполнения. Запросы к материализованному представлению могут затем пропускать заданное значение параметра, точно так же, как в кэше. Когда они это делают, база данных внутренне заполняет отсутствующее состояние, прежде чем ответить к приложению. Если приложение позже выполнит тот же запрос, результат будет сохранен в кэше. Со временем база данных удаляет результаты, к которым редко обращаются, чтобы сэкономить память и избежать накладных расходов на поддержание результатов, которые больше не интересуют приложение.

Подробная реализация данного алгоритма была взята из Partial State in Dataflow-Based Materialized Views, Jon Ferdinand Ronge Gjengset [4] и переработана для графовой базы данных, так же пришлось упростить модель запросов и сейчас она отвечает на запрос выдающий список сотрудников.

Протестируем эту модель точно также на растущих деревьях, постепенно добавляя те же самые вершины, что и в сравнении обычных запросов и в наивной модели.

Здесь же мы будем вызывать запросы достаточно часто (через каждые 10 добавлений в первом тестировании и каждую добавленную вершину для второго тестирования) и смотреть на обновление частично материализованного представления. В первом случае запрос тратит время только на обновление частично материализованного вида из кэша, что даёт ему значительное преимущество так, как он уже практически знает ответ на вопрос, таким образом среднее время ответа на запрос составило 121ms. Во втором же случае у нас каждый раз добавляется новая вершина, но нам не надо обрабатывать старые, т.е. мы задаём вопрос только о новой, таким образом среднее время ответа на запрос составило 58ms.

Как видим с помощью данного ускорения и последовательной загрузки документов мы смогли достаточно сильно ускорить запросы к нашей базе данных. Особенно при добавлении новых вершин, которые долго обрабатываются, как можно было видеть ранее при тестировании.

Заключение

Как видно такой подход к хранению официальных документов и обработке запросов для документов одной компаний приносит хороший выигрыш по времени работы, а это значит адаптация других компаний и других задач может быть ценным коммерческим продуктом, так как скорость и ресурсозатратность в нашем мире играют большую роль.

Ещё это стало небольшим шагом к созданию автономной системы создания графа по грамматике для официальных документов, так как мы показали, что это дает свои бонусы, а эта система помогла бы многим компаниям тратить меньше ресурсов на ведение документации. И в идеале человек не разбирающийся в программировании мог бы задать грамматику и загрузить архив документов, а на выходе получить набор запросов на которые он мог бы получать практически мгновенные ответы.

Список использованных информационных материалов

[1] Verifiable Credentials Data Model v1.1, W3C Recommendation, 03 March 2022, URL https://www.w3.org/TR/vc-data-model/

[2] JSON-LD Articles and Presentations, URL https://json-ld.org/learn.html

[3] Matrix Graph Grammars, Pedro Pablo Perez Velasco, February 2009

[4] Partial State in Dataflow-Based Materialized Views, Jon Ferdinand Ronge Gjengset, February 2021

[5] Amozon Neptune User Guide, URL https://docs.aws.amazon.com/neptune/latest/userguide/graph-get-started.html