МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Институт (Филиал): № 8 «Компьютерные науки и прикладная математика»

Кафедра: 806

Группа: М8О-406Б-21

Направление подготовки: 01.03.02 Прикладная математика и информатика

Профиль: Информатика

Квалификация: **бакалавр**

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

на тему: «Создание аналитического профиля клиента экосистемы в режиме streaming»

Автор ВКРБ:	Овчинников Дмитрий Максимович	(_____)
Руководитель:	Ненахов Евгений Валентинович	(_____)
Консультант:		(_____)
Консультант:		(_____)
Рецензент:		(_____)

К защите допустить

Заведующий кафедрой № 806 «Вычислительная математика и программирование» Крылов Сергей Сергеевич (_____)

Москва 2025

РЕФЕРАТ

Выпускная квалификационная работа бакалавра состоит из 56 страниц, 22 рисунков, 13 использованных источников, 1 приложений.

REAL-TIME, NEAR, REAL-TIME, KAFKA, CDC, CASSANDRA, GRAPHQL, АНАЛИТИК, КЛИЕНТ, ЭКОСИСТЕМА

Объектом разработки в данной работе является real-time/near real-time экосистема приложений для объединения и обработки данных из распределенных систем.

Цель работы – создание системы способной масштабироваться, обеспечивающей отдельную единую точку доступа к данным, минимизируя нагрузку на источники данных.

Для достижения поставленной цели были изучены современные подходы работы с потоковыми данными, инструментами CDC и оптимизации работы с запросами.

Основной результат работы — Backend инфраструктура позволяющая горизонтально масштабироваться и выполнять поставленные задачи.

Данные результаты разработки предназначены для использования компаниями крупного, среднего бизнесов, которым необходима актуальность и объединённость данных в аналитических запросах.

Применение результатов данной работы позволяет клиентам приложения удобно анализировать информацию, внедрять технологию сбора в текущие источники информации.

Содержание

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящей выпускной квалификационной работе бакалавра применяют следующие термины с соответствующими определениями:

Apache Cassandra	– распределенная NoSQL база данных
Apache Flink	– фреймворк для обработки потоковых и пакетных данных с высокой производительностью
Apache Kafka	– распределенная потоковая система для размещения, хранения информации
Backend	– серверная часть приложения, ответственная за обработку данных
Backpressure	– механизм управления потоком данных, позволяющих ограничивать их на стороне потребителя
Change Data Capture	– метод для захвата изменений в источнике данных и передачу их в другие системы
Debezium	– инструмент CDC для отслеживания изменений в базе данных
Eventloop	– цикл обработки событий, управляющий асинхронным поведением
GraphQL	– язык запросов, позволяющий точно указать необходимую структуру данных

gRPC – фреймворк Google для общения сервисов, использующий ProtoBuf

Histogram – тип метрик, предоставляющий значения в заданных интервалах

Java – объектно-ориентированный язык программирования с автоматическим управлением памяти

LSM-дерево – структура данных позволяющая оптимизированно хранить большие данные

PQL – язык запросов для Prometheus

Producer – элемент системы, создающий новые данные

Prometheus – база данных для хранения метрик с временными рядами

ProtoBuf – система сериализации данных от Google

Real-time – обработка данных в режиме реального времени с минимальной задержкой

TCP – протокол передачи данных с гарантированной доставкой и контролем ошибок

Zookeeper – сервис для координации кластера Kafka

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей выпускной квалификационной работе бакалавра применяют следующие сокращения и обозначения:

ACK – Acknowledgment

API – Application programming interface

CAP – теорема о выборе базы данных, основываясь на параметрах согласованности, доступности, устойчивости

CDC – Change data capture

LSM – Log-structured merge

OLAP – анализ больших данных в реальном времени

OLTP – система для обработки транзакций в реальном времени

RAM – оперативная память компьютера

TSDB – база данных, оптимизированная под хранение временных рядов

URL – стандартизированный адрес ресурса в сети

ВВЕДЕНИЕ

Актуальность темы связана с тем, что с развитием компании неизбежно растет и количество данных, которые приходится обрабатывать. Создаются отдельные команды со своей зоной ответственности. Зачастую они имеют отдельные распределенные источники информации, доступ к которым сильно ограничен внутри компании. Однако, с течением времени необходимо анализировать текущие тенденции. В результате на плечи аналитика выпадает задача об анализе работы текущих решений. Как итог возникает несколько проблем.

Он должен запрашивать доступ к каждой подобной системе.

Ему интересны настоящие данные, а значит необходимо затронуть продовольственную среду. В связи с этим, возникает дополнительная нагрузка на систему, что может приводить к временной приостановке работы баз данных, что скажется на недоступности к потребителю услуг.

Может существовать сотни таблиц откуда можно извлечь информацию, а также много отдельных топиков событий Kafka. Задача каждый раз выбирать как соединять все эти данные в ручном режиме слишком трудозатратна.

Из этого следует, что в определенный момент с ростом экосистемы компании поток данных необходимых обработать и соединить возрастает до такой степени, что аналитик перестает видеть полную картину, а также полученная информация на момент сбора будет не в актуальном состоянии. Плюс без автоматизации затруднительно добавить отдельный сервис, анализирующий в моменте потребности потребителя.

Из этого следует, что для решения подобная автоматическая система должна поддерживать следующие принципы:

- быть простой для расширения, т.е набор изменяющихся требований не должен значительно тормозить скорость разработки,
- быть высоко производительной, т.е при характерных больших запросах от аналитиков на наборе обширных и постоянно растущих данных, система должна минимизировать наличие узких мест производительности,

- обеспечивать бесперебойную работу системы, т.е шанс потери ценных данных при инцидентах должен быть сведен к минимально возможному,
- предоставлять простой интерфейс для конечного пользователя, т.е клиент, использующий систему, должен максимально просто обращаться к сложной логике работы системы.

Целью данной выпускной квалификационной работы является разработка аналитического профиля клиента экосистемы на основе потоковых данных клиентов различных индустрий экосистемы в режиме streaming processing. Нужно подготовить необходимую инфраструктуру для достижения масштабируемости, высокой степени надёжности и доступности. Применить фреймворк Flink[1] для обработки данных в real-time и near real-time режиме.

Для достижения поставленной задачи будет разработан и настроен комплекс инфраструктуры, который обеспечит выполнение обработки потоковых данных в real-time и near real-time режиме.

Подводя итог, задачи выпускной квалификационной работы:

- использовать технологии streaming для удовлетворения нагрузки на систему,
- получить метрики о работе конечного решения.

Взаимодействие с системой подразумевает индикацию клиенту о необходимости обновить конкретные данные в виде сообщений в топике Kafka.

Использование результатов работы позволит компаниям удобно интегрировать решение в текущий рабочий процесс. Позволит держать актуальные и объединенные данные с единой точкой доступа. Обеспечит масштабируемость под рост клиентской базы компании.

1 Теоретическая часть

1.1 Kafka

Кафка – это система, реализующая в себе принцип журналирования информации.

1.1.1 Почему без нее нельзя?

Представим, что мы разработаем сервис с открытой во внешний мир API. Ключевой задачей для нас будет – не терять данные, которые отправляются на этот интерфейс. В Java [2] приложении, являющейся наиболее распространенным языком в бизнес сфере, возможны две модели взаимодействия с запросами.

Стандартный, где мы на каждый запрос создаем свой поток, который и выполняет задачи, проблемой тут является, что если начинается выполнение синхронной операции, то поток будет занимать ресурсы приложения.

Второй вариант – использование project reactor и других реактивных библиотек. Где запросы обрабатывает легковесный eventloop поток. Тогда мы можем сохранить информацию о запросе и при выполнении задачи – вызывать callback функцию, в результате которой eventloop вернет ответ пользователю. С этим подходом мы создаем больший буфер для обработки, однако мы не застрахованы от следующего случая. Представим, что наше приложение работает на отдельной вычислительной машине. Однако, для экономии ресурсов, не только наше решение будет на нем запущено, а следовательно, в какой-то момент может закончиться оперативная память, либо будет произведена плановая замена оборудования. В этой ситуации все наши запросы, которые не были обработаны, будут полностью сброшены. В результате мы потеряем ценную информацию.

Для глобального решения проблемы необходим механизм backpressure. Глобально, мы хотим совместить два паттерна программирования. Это Iterator – который позволяет при нашем запросе запрашивать новую информацию из источника. Однако тут не ясно что делать если еще нет новых данных.

А также второй паттерн – observer. Его суть заключается в подписке клиента на источник информации. В результате при поступлении нового события мы сможем его обработать.

Вернемся к backpressure. Совместив указанные выше два паттерна, мы получим следующее условие – мы должны на стороне клиента контролировать нагрузку, которую мы готовы в данный момент обработать, чтобы не перегрузить приложение частыми запросами.

Тут и приходит на помощь предварительное хранилище в виде Kafka.

1.1.2 Особенность записи в Kafka

Так как эта система стоится на основе журналирования, то это означает, что мы должны записывать новую информацию только в конец. Если смотреть на конкретную реализацию, то мы можем увидеть, что внутри Kafka существуют очереди сообщений, в которые пишет отправитель. Однако с его стороны он пишет лишь в логическое разделение очередей – топик. Глубже – у каждого топика есть части – partition. Внутри такой части будет сохраняться порядок сообщений, на конкретной машине в кластере Kafka – брокере, может находиться такая partition. Однако физически она состоит из еще более мелких разбиений, называемых сегментами. В своей сущности это обычные файлы, как только мы записываем информацию, то в этот файл в конец записывается нужное сообщение. При том при достижении определенного размера сегменты закрываются и их нельзя будет менять. Запись ведется только в еще не заполненные полностью сегменты.

Как итог мы получаем главное преимущество – теперь наши запросы будут не в оперативной памяти, а храниться в этом временном хранилище находящимся уже на жестких дисках, т.е. обеспечивая сохранность уже сохраненного на долгий период времени.

1.1.3 Масштабируемость Kafka

Предположим, что мы положили сообщение в очередь, оно успешно сохранилось и ждет пока мы его дальше прочитаем. С точки зрения абстракции – мы встречаемся с сущностью consumer. Главной особенностью в этой

системе является, то, что он способен вычитывать сообщение более одного раза, если захочет. Внутри Kafka лежит отдельный топик в котором будет находиться номер указателя для конкретной консьюмер группы.

Где нам это может пригодиться? Представим, что приложение клиент упало и не смогло обработать последний запрос. Но для нас важно обработать всех. Тогда для нас доступны механизм уведомления. Рассмотрим способы, которые нам доступны.

Во-первых, как только мы получили сообщение, то мы можем отправить этот сигнал обратно, чтобы указатель в очереди стал на следующий элемент. Однако если мы завершили выполнение программы неуспехом, то мы не сможем вернуть указатель обратно, ему доступно только инкрементирование.

В тоже время существует второй вариант, когда мы только начинаем обрабатывать сообщение и после завершения будем отправлять успех. Однако заметим, что, неужели, нам придется создавать нового клиента, если мы захотим поднять вторую сущность приложения клиента? Оказывается, создатели Kafka продумали этот момент заранее.

Для этого существует consumer group. На нее существует общий указатель, и если ты тот, кто читает сообщения, то можешь использовать тот же идентификатор группы. Группа работает с количество партиций. Если, к примеру их три, и один получатель, то он и будет обрабатывать все три партиций самостоятельно. Представим, что к нему присоединяется еще одно приложение с такой же группой, тогда будет произведен механизм пере балансировки, в результате которого эти три группы распределяется на две, а именно по две и одной партиций для таких слушателей. Если же один из клиентов не способен отвечать оговоренное время, то Kafka исключит его из группы и отдаст не обработанные сообщения прошлого консьюмера тому, кто остался активным. Как только количество клиентов в своей группе становится больше, чем партиций, то те, кому не хватило – останутся без партиций, поэтому при масштабировании на боевой стенд нужно учитывать количество партиций.

1.1.5 Поговорим о гарантиях для отправителя

Представим, что мы готовы отправить сообщение, однако даже начав передачу информации по сети у нас нет никаких сведений о том – а жива ли сам кластер Kafka? Такое возможно, если был провал сети из-за аварии и сетевые запросы не дошли до получателя, либо сама Kafka может производить выбор нового лидера партиций, в результате чего быть краткосрочно недоступной.

Чтобы быть точно уверенными, что мы положили нужное сообщение на сторону Kafka, существует настройка acks. Рассмотрим каждый из вариантов.

Стандартное поведение это acks равный нулю. В таком случае мы просто отправляем сообщения в надежде, что на другой стороне нам их поймают и обработают. В результате работаем быстро, но ничего не гарантируем.

Второй вариант acks 1. С такой настройкой мы способны получать подтверждение от лидера о успехе записи на него. В результате в большинстве случаев мы хотя бы доставили до Kafka сообщения, но не решили дополнительной проблемы. А именно это падение лидера. Это может возникнуть по разным причинам т. к. обычно разные брокеры находятся на разных машинах и возможно что-то пошло не так на главной. В результате наши реплики должны выбрать нового лидера. Однако так как лидер не заботился о своих копиях, то не факт, что все сообщения у них будут, которые отправляли.

Настройка acks -1 позволяет удостовериться, что лидер будет ждать чтобы копии получили точно все сообщения, которые прислали лидеру. И в ответ отправили ответ ему же об успехе, в результате пока мы это ожидаем – мы не сможем принимать другие сообщения и кидать исключение о процессе сохранения информации. Этот вариант является относительно долгим по сравнению с другими двумя, однако в случае аварии мы сможем точно гарантировать, что все что обработал лидер гарантом будет и лежать в копиях.

1.1.6 А сколько же нам нужно копий?

В реальности если мы будем ждать абсолютно всех, то мы превратим наш процесс в полностью синхронный т. к. пока все не ответят мы не можем продолжить работу. В большинстве случаев компания выбирает следующий вариант. Мы говорим, что только одна из реплик может быть ведущей. Т.е. количество подтверждений для лидера будет всего единицей остальные сущности будут асинхронно подбирать данные в фоне.

Как итог при аварии у нас будет точно на кого переключиться, но в тоже время и практически готовые реплики по соседству. Говорят, что мы способны выдержать N-1 падение, где N это количество реплик с синхронным подходом. Важно понять, что для сохранения консистентного состояния в момент сохранения для полной гарантии мы должны не принимать другие запросы на запись. Поэтому тот, кто будет писать в Kafka получит ошибку о текущем не готовом к работе состоянию.

1.1.7 Почему модель с ведущей сущностью выиграла, ее альтернативы

Если мы рассмотрим случай, когда мы хотим использовать другую модель, то мы придем к такой, где каждая из сущностей является главной. Это означает, что из-за одинаковых ролей мы можем написать в любую из них. У этой системы есть значительная проблема.

Представим, что мы записали в первую сущность, когда мы сможем получить вторую, которая сможет быть такой же как первая? Для ответа на этот вопрос мы должны обеспечить связь между ними. Если мы создадим виртуальное пространство и запустим их несколько на одной машине, то при падении такой, мы не защитимся ни от чего.

Другой вариант — это использовать разные сервера с отдельными запущенными процессами. Но так как мы соединим их сетью, то у нас могут возникнуть проблемы с ней. К примеру, в ней были скачки аварии, в результате которых сообщение потерялось. Как итог мы привели сущности в разные состояния.

Как можно бороться с таким поведением? Мы можем добавить еще один узел. Если между двумя ломается связь, то добавив третий возможно связь между основными будет проходить именно через него. Однако мы ничего не решили, между ними все еще лежит сеть, она может быть сломана, нужны еще сервера промежуточные. И как итог мы затрачиваем значительное количество усилий и денег для поддержания большой системы, которая не обеспечит для нас устойчивости в условиях аварии.

1.1.8 pull, push модели для репликации

Мы поняли, что прошлая модель взаимодействия не подходит и мы натыкаемся на значительную проблему записи, которая ставит на нет работу с важной информацией. Подход, который все же применяется это pull либо push модель для взаимодействия с основной сущностью. Рассмотрим, когда мы делаем push в реплики. Легко заметить, что как только на нас отдается такая проблема, то мы хотим все же получать подтверждение о том, что нам не удалось это сделать. Т.е. проблем стало не меньше, однако более решаемо. Мы можем ждать подтверждение не от абсолютно всех реплик, потому что будет иначе применен синхронный

Второй вариант это pull. Т.е. реплики будут по факту являться потребителями, запрашивая информацию от актуальной версии. И через какое-то время хоть мы и не сможем гарантировать абсолютно все записи в этих сущностях, однако до заполнить всю информацию будет проще, т.к. мы не обязаны проходить полностью по очереди с самого начала. Это позволит создать сущность, работающую в асинхронном подходе.

1.2 Cassandra

1.2.1 Выбор базы данных

В пользу выбора определенного хранилища информации стоит рассмотреть CAP теорему. Она позволяет понять какие качества мы хотим увидеть от базы. При том утверждается, что мы можем получить только два из них в конкретной реализации. Выделим эти характеристики:

- Согласованность. Все ноды должны иметь актуальные данные;

- Доступность. Если происходит сбой, то мы все равно должны обработать запросы;
- Устойчивость к разделению. Если связь пропадет между нодами, то мы продолжим взаимодействие.

Рассмотрим куда входит типичная реляционная база данных PostgreSQL. Можно выделить CA из этих определений. Мы действительно можем обеспечить доступность и согласованность, однако, когда сеть становится недоступной, то мы должны ждать соединения с нашей репликой. До этого момента система ожидает в блокирующем состоянии, пока мы не станем снова на связи.

Вторым примером станет Cassandra. Проанализировав ее поведение, приходим к выводу, что это AP система. В процессе репликации используется аналог сплетен. Те, кто знает актуальную информацию сообщает об этом ближайшим соседям. Это походит на распространение вируса в эпидемию, но наша информация, наоборот, несет только пользу.

Из этого можно сделать выводы о том, что повлияло на такое определение.

Синхронная репликация блокирует вызов, позволяя достичь согласованности, однако это требует времени, поэтому снижается доступность.

Асинхронная репликация позволяет системе дольше находиться доступной, однако несогласованность может привести к удручающим последствиям, если мы используем напрямую ценную информацию.

Из этих двух миров мы нашли минусы в каждом, хочется увидеть более идеальное решение. Таким обладает Kafka. Хоть она и хранит события, но все же сохранением на время информации похоже на базу данных. Она использует подход, когда несколько реплик остаются согласованными с главной, а остальные забирают информацию. Это позволяет содержать минимум N реплик готовых заменить основную, но в случае, когда останется одна актуальная, то придется перейти к синхронному подходу.

Из этих рассуждений следует, что задача выбора базы данных лежит на плечах архитектора системы, он должен заранее продумать взаимодействие, чтобы обеспечить предложенные свойства системы. Основываясь на ней, выбираю Cassandra для дальнейшей работы в системе.

1.2.2 OLTP vs OLAP

Стоит заметить, что в привычном понимании мы можем работать с двумя типами базы данных. Первая получила название OLTP т.е. обработка транзакций в реальном времени. Обычно приложение будет искать в такой базе данных лишь небольшую часть записей, используя индексирование. Однако для целей данной работы нужно значительно расширить возможности обработки данных.

Тут подойдет второй подход, а именно для аналитической обработки данных в реальном времени, или по-другому OLAP. Такое решение имеет ряд особенностей, отличающихся от первого решения. Простой аналитический запрос скорее будет смотреть на определенную колонку, чтобы получить нужную информацию, чем искать отдельные небольшие записи по ключу. К примеру, запросы могут быть такие:

- какая общая прибыль за период августа по продажам автомобилей,
- насколько больше было продано кредитов за период этого месяца, чем за такой же период прошлого года,
- какую марку товаров в магазине покупатели чаще выбирают и покупают.

Вероятнее всего, что подобный запрос поможет аналитику вставить важную информацию в инфографик, чтобы оценить принятое решение, либо скорректировать на такой тенденции новые возможные планы.

1.2.3 Хранение данных в Cassandra

Важно понимать, что подобная информация может значительно превышать хранилище обычной базы данных, предназначенной для взаимодействия с пользователем. Можно смело предположить, что аналитик

будет взаимодействовать с терабайтами, или даже петабайтами информации, в то время как в обычном решении мы бы вероятно хранили гигабайты или терабайты информации. Стоит заметить, что в таком складе данных может храниться история состояний, на основе которых можно за короткий период выделить тренды и скорректировать бизнес-модель, чтобы показать пользователю более релевантную информацию.

Почему нам не стоит производить всю работу в одном месте?

Во-первых, стоит отметить, что запросы могут идти минуту и более, если мы запросили слишком много информации, в условиях

За счет чего же можно хранить такой объем информации? Отличительной особенностью Cassandra является ее способ хранения информации на дисковом пространстве. Если мы рассмотрим обычную реляционную базу данных, то увидим, что информацию располагают построчно. В этом есть свои плюсы т. к. найдя нужный id мы можем сразу узнать полезную информацию. К примеру, хранить основную информацию о профиле клиента. Но что делать, если нужно хранить рядом много информации? Например, блок о себе, тогда подойдет база данных документ ориентированная. В таком случае мы храним не по строкам, а в виде непрерывной последовательности байтов. Это означает, что найти нестандартизированного размера информацию будет быстрее.

Однако вернемся к изначальному хранилище реализующему стратегию хранения столбцов. Во-первых, сталкиваемся с проблемой выбора нужной строки, однако она легко решается, необходимо всего взять нужный индекс и забрать то, что лежит под ним из каждого столбца.

Одной из возможных оптимизацией для сжатия будет являться натура данных, которые мы собираемся хранить. К примеру, если это хранилище товаров для определённой группы людей, то количество различных групп может быть значительно меньше самих записей, иначе говоря, не все значения в столбце уникальные. К примеру, разбиение на мужчин и женщин. Тогда эту информацию мы можем закодировать как ноль и единица. После чего

превратить наш столбец в последовательности нулей и единиц, тем самым, не храня полную строку о с такими названиями, однако имея возможность декодировать полученное хранения в удобный для чтения вид. Но это не полная оптимизация. Предположим, что эти нули и единицы могут достаточно часто идти подряд. Тогда мы можем записать последовательность из десяти нулей, дальше пяти единиц, и после восьми нулей, как строку 10, 5, 8. Это значительно сократит количество информации, которое мы будем хранить. Таким образом и работает кодирование с помощью битовой маски.

1.2.4 Индексирование

С точки зрения пропускной способности самый быстрый способ записать данные, это записать их в конце. То есть нам желательно записать именно в это место при добавлении записи в наш столбец, однако для запросов агрегации вероятно, что мы захотим искать информацию в отсортированном виде. К примеру, отсортировать продажи по месяцам. Важно заметить, что сортировка по всем столбцам по отдельности не будет иметь смысла, мы не сможем по индексу восстановить полную строку, которая была изначально, поэтому оператор базы данных может самостоятельно на основе данных, которые будут храниться выбрать наиболее важные для него столбцы, по которым можно будет искать в диапазоне информацию. Таким образом можно сделать вторичные индексы, которые будут хранить указатели на нужные строки, однако из-за сортировки будет быстрее происходить поиск информации. Также если мы вводим сортировку, то сжатие с битовыми масками будет работать еще лучше, позволив сжать таблицу из терабайтов в килобайты, если она достаточно разряженная.

1.2.5 Запись в столбцовое хранилище

В обычной реляционной базе данных при записи мы стараемся сразу сохранить информацию в месте, где она должна быть. С такой задачей прекрасно справляются B деревья, позволяя вставить информацию в середину.

Однако, для столбцовых хранилищ в отсортированном виде придется переписывать все файлы столбцов, так как вставка должна их обновлять согласованным способом.

Правильным решением будет использовать LSM деревья.

В них используется такой принцип – мы храним самую важную информацию в оперативной памяти, таким образом мы получаем доступ к быстрой памяти и можем записать туда информацию. При накоплении информации, так как нет RAM способной хранить огромный пласт информации – они объединятся с файлами столбцов на диске и запишутся блоками в новые файлы.

С точки зрения аналитика мы не будем делать подобные сложные запросы, где нужно проверять несколько мест для данных, однако оптимизатор запросов будет знать о такой реализации и скрывать от конечного потребителя устройство сложной системы.

1.3 GraphQL

1.3.1 Как выбрать модель взаимодействия

Несмотря на то, что у нас уже есть хранилище информации, в которую мы стягиваем извлеченные данные с помощью предобработки мы должны обеспечить клиентам удобное взаимодействие с этой информацией.

Для решения такой задачи введем промежуточное приложение способное справиться с поставленной задачей. Представим путь предоставления данных. Клиент запрашивает необходимые колонки, приложение ходит в хранилище для получения актуальной информации, и отправляет в удобном формате обратно. После получения данных приложение десериализует их в нужный формат. Таким образом превращая удобное представление данных программы в последовательность битов, которые можно отправить через сетевое соединение в дальнейшее место назначения.

За историю развития компьютерных технологий были придуманы разные контракты позволяющие решить такую задачу. Принимающая сторона,

получив последовательность битов должна заниматься из декодированием, то есть переводом обратно в удобный для использования формат.

1.3.2 Средства языка программирования

Самый простой способ — это воспользоваться стандартными средствами языка программирования, который и получает из хранилища данные. Однако, стоит заметить, что мы становимся в положение, в котором затруднительно использовать сторонние языки программирования, так как выбираем специфичный путь для данного. В результате на принимающей стороне должен быть еще один сервис промежуточный способный вычитать необходимый для нас ответ.

Также если мы работаем с ценной информацией, то необходимо обеспечивать безопасность такого соединения, однако из-за особенностей языка и его не совершенности возможно мы сможем удаленно запустить вредный код, послав определенного вида запрос.

Возникают и проблемы добавления новой информации в контракт, он должен быть согласован, чтобы предоставляемые классы спокойно могли кодироваться и декодироваться.

Также ставится вопрос производительности подобной системы. К примеру, в языке Java размер полученной структуры может быть настолько большим, что это будет занимать значительное время приложения, став узким местом производительности.

Эти минусы ставят на нет долгосрочную перспективу использования стандартных библиотек языка программирования, однако если задача кратковременная, то это может стать быстрым решением, хоть и сомнительным.

1.3.3 JSON и XML

Рассмотрим другой способ общения приложений. Это JSON или XML.

В сравнении с предыдущим способом взаимодействия мы подходим к обратной стороне монеты с точки зрения удобного чтения людьми информации.

Формат JSON очень подходит для понимания пользователя. Особенную популярность этот формат получил благодаря встроенной поддержки в браузерах, однако и для стандартного взаимодействия между серверными приложениями такой формат часто выбирается разработчиками.

Мы могли бы остановиться на нем, но есть и особенности.

К примеру, в формате XML, и CSV невозможно различать число и строку. Такое поведение вполне вероятно приведет к проблемам при дальнейшей обработке данных. Эта проблема имеет решение, для которого нужно ввести внешнюю схему, которая позволяет явно задавать типы данных. Однако использование схемы добавляет дополнительную нагрузку на систему, требуя её постоянного обновления и синхронизации с изменениями в структуре данных, что усложняет поддержку и увеличивает вероятность ошибок. А JSON может различать числа и строки, но не целочисленные и значения с плавающей запятой. Эта особенность заложена в самом формате. Это может оказаться критичным, если мы используем в работе отслеживание высокоточных научных вычислений или данными с финансовыми составляющими, где важно корректно определять точность чисел.

Также мы ограничены размером чисел. В JSON есть проблема, что мы не можем представить число больше 2^{53} степени в двоичном формате из-за стандарта IEEE 754. Может показать не проблемой, но если будем передавать число из 64 бит, то придется его представить в виде строки, а не числа для решения такой проблемы.

Есть и проблема со строковыми данными. Формат JSON и XML умеет успешно работать с кодированием Unicode строками, но не двоичными. Если бы можно было представить все, как только второй формат, то можно было бы сэкономить больше полезной информации в меньшем объеме. Для обхода такой проблемы используют кодирование в base64. Этот формат может зашифровать строку в двоичный формат, однако его нельзя назвать настоящим шифрованием. Любой человек может превратить эта данные в исходные и

получить информацию. Но такой трюк позволяет хоть и обходным путем, но увеличить объем данных всего на 33 процента от изначальных.

В CSV формате схема отсутствует, это означает, что, получив данные в таком формате мы сами вольны интерпретировать их как захотим. Но это и минус, так как нам вручную придется дорабатывать логику, чтобы поддерживать внезапно изменившийся контракт (добавление нового столбца)

В формате XML и JSON есть поддержка схемы, но она не является обязательной. Само наличие такого функционала позволяет приложению производить самостоятельно кодирование и декодирование в заданные рамки, однако в противном случае неиспользования приходится переносить эту логику на уровень приложения.

Несмотря на все представленные минусы, эти форматы все еще пользуются огромной популярностью для обмена данными между приложениями. Так как обычно мы взаимодействуем не с абстрактным вендором готовым в любой момент меняться, а вероятнее всего с другой командой. Тут на передний план выходит возможность договориться между ними, чтобы поддерживать актуальный контракт на каждой из сторон.

Но такие проблемы не смогли остаться в стороне, и энтузиасты не сидели сложа руки, а занимались созданием более продвинутых технологий. На основе JSON появилось большое количество двоичных кодирований для него.

1.3.4 Двоичный JSON

Рассмотрим подобные альтернативы.

Одним из кандидатов является MessagePack.

Алгоритм его действия заключается в:

- указываем сколько будет идти объектов в первом байте,
- смотрим какой тип и какой длины, к примеру, строка длины 8,
- вычитываем эту строку, получаем имя поля,
- повторяем вычитывание типа плюс значение и получаем значение.

Таким образом мы закончили алгоритм и можем вычитывать из последовательности байт полезную информацию.

Однако, к примеру это может превратить строку из 81 байт в 66 байт. Что, несомненно, является плюсом, но мы теряем возможность быстро прочитать полученные значения человеку. Это может выливаться в проблемы при логировании. Мы сможем получить нужные взаимодействия, чтобы понять путь внутри микросервисов, однако команде сопровождения придется обращаться к помощи разработчиков, либо самостоятельно тратить больше времени на определение проблемного поведения.

Но люди не остановились на этой технологии.

1.3.5 Thrift и ProtoBuf

Для описания контракта между приложениями существует Thrift и Protocol Buffers. Эти библиотеки основаны на двоичном принципе кодирования, но требуют описания схемы для их работы. В коде это выглядит как авто генерация из полученного описания нужных классов.

Эта схема позволяет хранить данные в другом формате, если раньше пришлось бы закодировать имя поля, которое мы хотим передать, то при создании схемы мы можем присвоить полю нужный тег. Не трудно заметить, что если именем мы будем использовать строки, то мы получим существенный выигрыш при кодировании информации. Будет достаточно соотнести тег с схемой и получить оттуда нужную информацию. Таким подходом мы создаем псевдонимы для полей.

Не редко при запросе к системе необходимо валидировать входящие данные. К примеру удостовериться, что уникальный идентификатор был успешно расшифрован из вызова к сервису. Обычно такая логика переносится на плечи разработчика, который будет в начале обращения проверять это. Но схема позволяет ввести ключевые слова required и optional. Все просто – можно активизировать проверку во время выполнения запроса с помощью средств библиотек.

Поговорим о миграции. Описание схемы — это замечательный инструмент, однако, что же делать, когда нужно изменять контракт? Такое

изменения называется эволюцией схемы и желательно сохранять прямую и обратную совместимость.

Если тот, кто будет посылать запрос решит, что поле не должно быть задано, то все просто – так как тега не будет описано в запросе, то мы и не заполним данное поле. Однако из этого следует, что порядок полей важен. При их описании мы задаем последовательность номеров. Дальше без подстановки в таблицу мы не сможем успешно восстановить информацию. Поэтому мы можем менять название поля в схеме, но не желательно менять тег поля, так как это приведет к мусорным данным.

Для прямой совместимости при добавлении нового поля, описав его тегом, можно игнорировать его в старом коде. Таким образом мы можем работать в таком формате.

Для обратной совместимости – новый код сможет прочитать старые данные, так как теги из упоминания выше мы не меняем, однако новые поля должны быть не обязательными. Иначе мы бы спотыкались на проверку валидации такого запроса. Альтернативным решением может стать дефолтное значение нового поля, однако это не всегда подходит под бизнес требования к логике.

Удаление также возможно. Но это должно быть поле не обязательным, а также этот тег теперь не стоит использовать повторно.

Можно и использовать массивы. Для этого предназначено ключевое слово repeated. Можно менять optional на этот тип. Тогда новый код сможет увидеть пустой список для новых данных, а старый увидит последний элемент из списка.

При работе с числами есть возможность изменения области типов, но сразу стоит принимать во внимание, что расширить в большую сторону возможно приписав нули, а в обратную из-за меньшего количества возможных чисел в множестве можно перейти, убирая часть данных, тем самым возможно теряя изначальный смысл в заложенных данных.

Но по итогу посмотрев на предложенные варианты было решено выбрать GraphQL.

1.3.6 Контракт GraphQL

Какие же преимущества ставят этот вариант предпочтительным?

Во-первых, Open Source этого проекта позволяет находиться ему в среде поддерживаемых комьюнити решений, в результате количество багов становится меньше, из-за вовлеченности пользователей, а также возможна самостоятельная доработка решения, если требуется. Альтернативой для общения сервиса мы могли бы выбрать REST взаимодействие, однако у этого решения есть такие проблемы, как:

- Избыточность либо недостаток данных. Мы обычно можем получить либо больше, чем требуется, либо наоборот не все данные, как итог это вынуждает использовать несколько запросов, собирая по крупицам нужную информацию;
- По хорошему тону разработчиков – каждая точка запроса в приложение представляет собой ресурс, либо взаимодействие с множеством ресурсов. В GraphQL у приложения всего одна входная точка, а схема определяется на стороне сервера, поэтому клиентам проще запрашивать информацию, не храня кучу конечных URL;
- В работе с базой данных может возникать проблема N+1 запросов. Это означает, что на один наш большой запрос мы должны на самом деле делать дополнительные N запросов к серверу, чтобы собрать всю информацию. Это закономерно сказывается на количестве запросов к базе данных. В итоге мы получаем и так нагруженную долго работающую систему (из-за объема ее информации), а также кучу дорогостоящих обращений к ней. GraphQL позволяет в одном запросе выразить необходимые связи и решить эту проблему.

Благодаря чему же возможно описать такой запрос, где будет вся необходимая информация?

Ответ кроется в самом названии, так как структура запроса представляется в виде графа, где в узлах мы увидим объекты, а в ребрах сами связи между ними.

Существует несколько типов запросов, такие как:

- «Query». Это аналог GET для REST. Однако все типы запросов будут отправляться через POST запросы для такого соединения. Тут стоит заметить, что мы не ограничены REST, если в структуре микросервисов есть веб-сокеты, или gRPC, или другой транспортный протокол, то это не будет являться проблемой и GrahQL сможет работать и через них;
- «Mutation». С этим запросом можно изменять данные в базе. Это аналог POST и PUT в REST;
- «Subscription». Позволяет слушать изменения в базе данных в реальном времени.

Сами запросы также дополнительно помогают клиенту с выбором. К примеру, нам доступны такие инструменты, как: выбор поля, выбор аргументов для него, на который мы можем ссылаться. Можем самостоятельно переименовать поле в ответе, с помощью псевдонимами. Возможно разбить запрос на фрагменты, использовать именованные функции для упрощения финального запроса. А также давать директивы. Они позволяют, к примеру пропустить поле, если запрашиваемая переменная будет true.

Для работы всей этой магии необходимо иметь схему GraphQL находящуюся на сервере приложения. В ней описаны два объекта, это TypeDefs и Resolvers.

Первый определяет список типов, которые в целом доступны. Поддерживаются четыре типа данных, это String, Int, Float, Boolean. Если указать восклицательный знак, то поле будет обязательным.

Resolvers же является распознавателем данных. Соответственно это указания как соотнести поля с тем, что реально лежит под завесой работы приложения.

Из всех пунктов становится ясно, что мы получаем бонусы в виде:

- автоматической генерации схемы для API, что снижает сложность разработки,
- оптимизация запрашиваемых запросов,
- работу с стандартными операциями CRUD для работы с базой данных,
- адаптируемость, т.к разработчик может расширить контракт определив новую схему, тем самым позволяя пере использовать код и снижать его избыточность.

1.5 Flink

1.5.1 Потоковая и пакетная обработка информации

Определим, как можно обрабатывать информацию. Зачастую нам известен конечный размер данных. К примеру, обработать все колонки базы данных, получить список покупок для пользователя, выполнить сложную операцию над заданным набором чисел. Мы можем обработать это с помощью пакетной обработки данных. Зачастую подобные операции выполняются долго, и пользователь не всегда ожидает из завершения. Но возможно запускать по расписанию, через определенный промежуток времени обработку. Но в условиях системы почти реального времени необходим другой подход. Поточная обработка. В ней мы реагируем на события и сразу их обрабатываем.

Часто набор данных не ограничен и постепенно поступает на вход. Реальные пользователи не желают ждать выгрузки необходимой информации раз в час, а то и раз в день. Можно обойти такие вводные с помощью создания искусственных промежутков для поточной обработки, но по мере уменьшения интервалов мы придем к мгновенной обработке.

В типичном случае мы бы считывали целые файлы раз в промежуток, однако аналогов в поточном варианте будет событие, или по-другому минимальная единица произошедшего в текущий момент. К примеру, пользователь положил в корзину товар. Зачастую нам важна временная метка подобного события.

Такие события можно хранить в Kafka, а дальше разные потребители могут обработать их.

В системе Flink возможна работа как с потоковой обработкой, так и пакетную обработку. Более подробную информацию о различиях подходов можно найти в книге Клеппмана о высоконагруженных приложениях [3], а также возможностях Flink в книге Уэске [4]

1.5.2 Внутренние особенности составляющих Flink

В кластер входит Job Manager и Task Manager. Первый координирует работу, а второй может ее выполнять. Они работают в связке, рассмотрим, как можно получить сообщение.

Здесь появляется концепция источника данных.

Он состоит из трех элементов:

- «Split» (Разделение). Эта часть данных, которую мы хотим обработать;
- «SourceReader». Тот, кто будет запрашивать разделения. Для каждого Task Manager будет создан свой считыватель, из этого следует возможность распараллелить поток;
- «SplitEnumerator». Элемент, отвечающий за распределение нагрузки, поддержание нагруженности читателей. Он находится в Job Manager, поэтому необходим один экземпляр для разбития ресурсов.

Инструмент позволяет также записывать в необходимые места информацию, к примеру базу данных Cassandra.

Однако простым перекладыванием мы не ограничиваемся. Между этими событиями возможно использовать отдельные операторы. К примеру, преобразовать данные в другой формат, отфильтровать их по заданным параметрам. Но такая цепочка вызовов не ограничена линией. Можно разделять поток на несколько операторов, соединять во едино. Такая гибкость позволяет пред обрабатывать данные в нужный формат.

Стоит отметить, что помимо предложенных функций мы не боимся и скорости обработки информации.

В типичной работе приложения часть операций может занимать больше времени, чем другие. Отсюда стоит заметить, что ничего не мешает использовать параллельное исполнение нескольких операторов, находящихся на разных этапах цепочки.

1.5.3 Backpressure

Для реализации используется механизм backpressure.

Подобная технология используется и в других ситуациях, к примеру протокол передачи информации TCP в себе имеет возможность контролировать поток информации.

Обратимся к абстракциям. Существуют общепринятые паттерны программирования. Это сбор определенных решений, которые люди записали в общее место, чтобы при возникновении типичных проблем обращаться к готовому ориентировочному решению. Один из таких паттернов — это итератор. Принцип прост, если мы имеем ограниченный источник информации, то мы готовы предоставить способ прохождения по нему. Здесь таким исходным местом может являться база данных, структура данных, то есть все что можно описать конкретным проходом по коллекции элементов. Однако не ясно как следить за новыми данными и как по ним перемещаться. Ведь обычно их конец означает остановку прохода.

Однако такую проблему решает второй паттерн программирования, observer. Этот наблюдатель с первого взгляда не имеет ничего общего с первым. На источник информации мы создаем сущность способную реагировать на поступающую информацию. Таким образом мы создаем callback. То есть указание реакции на запрос.

Люди через время заметили, что по своей сущности мы смотрим на одну картину. Есть откуда и куда. Плюс взаимодействие прямое и обратное.

Таким образом появилось backpressure. Мы хотим перемещаться по входящим элементам как с помощью итератора, но также и реагировать на новые.

Как же это работает на самом деле?

Программа, когда понимает, что не может одновременно выделить ресурсы на новый элемент потока – отказывается его принимать. А как только появляется возможность его обработать, то запрашивает новое из источника.

1.5.4 Интеграция backpressure в Flink

Таким образом, возвращаясь к Flink. Операторы работают независимо, реализуют backpressure, благодаря чему, способны не ждать друг друга и накапливать в буфере обработанные элементы.

Если ориентироваться на реальный пример, то со стороны наблюдающего можно будет заметить, как приложение нагружено на полную мощность, как только прекращает текущую задачу и становится только на тридцать процентов загруженным, тогда накопившиеся данные сразу заполнят доступное место.

Но у такой системы есть значительный минус. Представим, что авторы действительно закончили на этом обработку. Для мгновенной реакции приложения мы храним эти частицы потока в оперативной памяти. Таким образом, когда наше приложение падает, то мы сталкиваемся сразу с двумя проблемами:

- мы не способны вспомнить в каком операторе что находилось, а значит теряем данные,
- нам нужно знать, а в каком состоянии мы изначально были, чтобы при поднятии приложения не делать с начала обработку заново.

Для второго пункта уже помогает Kafka, так как позволяет хранить в себе отдельное смещение для потребителей. При прочтении новой информации можно отправить подтверждение обратно в кластер. Тогда мы запомним, где уже обработанная часть.

Kafka позволяет не отправлять подтверждение сразу, а к примеру на финальном этапе взаимодействия. Но в этом случае процесс синхронный, так как будем получать тот же элемент если запросим еще раз.

Тут помогает разобраться Flink. Он использует другой паттерн программирования – снимок. Суть простая, мы имеем возможность сохранить

состояние в постоянное хранилище, а также забрать оттуда информацию. С такой реализацией при преждевременном завершении приложения Flink будет способен загрузить контрольные точки для операторов и продолжить в том же месте, где он закончил в прошлый раз.

Стоит отметить, что дальше цепочку операторов возможно дуплицировать на другие Task Manager. Тем самым увеличив пропускную способность приложения. В типичном варианте расширение происходило бы с помощью горизонтального увеличения количества реплик, однако очень удобно, что можно настраивать коэффициент параллелизма, оставляя за главного Job Manager.

1.6 Метрики и логи приложения

1.6.1 Зачем необходимы метрики

В современном мире сложно в реальном времени оценить работу приложения.

К примеру, для java приложения мы сразу можем иметь косвенные показатели приложения. Потребление процессора, а также оперативной памяти. Но зачастую логика приложения настолько запутанная, а нагрузка большая, что эти параметры не сильно релевантны. Они хороший показатель для определения, что проблема существует в целом, но дальше бесполезны.

Также возможности Java позволяют делать снимки памяти, чтобы понять, что происходило, однако это все еще трудоемкий процесс, в котором нужно разбираться отдельно. Стоит отметить, что далеко не все создатели языка программирования, инструмента, фреймворка закладывают возможности мониторинга в реализацию.

Но тут на помощь приходит Prometheus.

1.6.2 Prometheus

В своей сущности это отдельная база данных, способная хранить в себе информацию. Но с отличительной особенностью.

Мы должны хранить не строки или столбцы, а метрики приложения.

Такими показателями могут быть счетчики внутри приложения. К примеру, количество вызовов внутри кода для определения нагрузки в приложении.

Хранить предлагается цепочки числовых событий. То есть нам нужен уникальный идентификатор для такой последовательности. В дальнейшем можно проанализировать как она менялась с ходом времени.

Отличительной особенностью является простота добавления для разработчиков. В самом коде достаточно предоставить адрес, по которому будут доступны метрики в конкретный момент времени.

А сами показатели это всего лишь текст. Стоит отметить, что адрес может содержать систему папок, где, дописав название в конец мы попадали бы на отдельные метрики.

Соединив воедино информацию, мы получаем, что не нужно хранить данные, не нужно их пред обрабатывать, лишь добавить простого агента.

Из реализации ясно зачем нужны цепочки в хранилище, по-другому мы не поймем чьи и откуда эти были метрики, чтобы не запутаться.

Отличной возможностью с такими стандартизированными выходными данными является объединение метрик разных приложений, написанных на разных языках. Таким образом реализуя библиотеку единожды для преобразования в нужный формат, мы получаем кластер метрик.

Также как и обычная база данных мы можем делать отдельные запросы на специальном языке запроса метрик PQL.

Рассмотрим основные возможности.

Во-первых, Prometheus [5] сам может делать pull для данных в прописанных точках. Обычно в связке с Kubernetes эти адреса расшифровываются с помощью service discovery.

Хранение происходит в TSDB – это как раз хранилище временных рядов.

На основе полученных данных можно настроить отправку уведомлений для команды сопровождения. К примеру, получать на почту, что происходит атака на сеть.

1.6.3 Типы данных в Prometheus

Рассмотрим типы данных, которые мы можем достать в этой системе. Первое – Gauges, приведено на рисунке 1. Этот тип является датчиком.

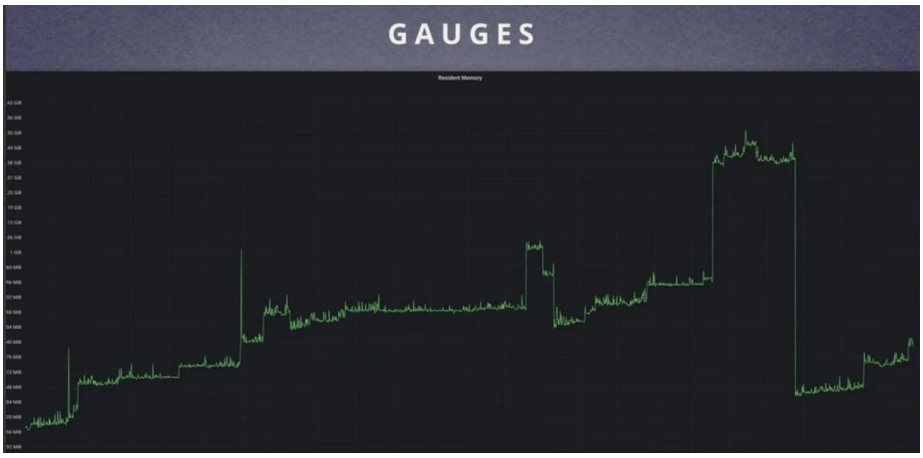


Рисунок 8 – Пример работы датчика в Prometheus

Он может хранить в себе число. При том оно может как расти, так и понижать свое значение. Полезно если мы хотим знать количество запросов.

Следующий тип – Counters(счетчики), приведено на рисунке 2.

Рисунок 2 – Пример работы счетчика в Prometheus

Как видно из графика линия растет только вверх. Тип похож на первый, но с урезанным функционалом. Подобный график можем увидеть, если хотим хранить количество событий. Например, сколько было продаж.

За ними следует Summaries (суммы), приведено на рисунке 3.

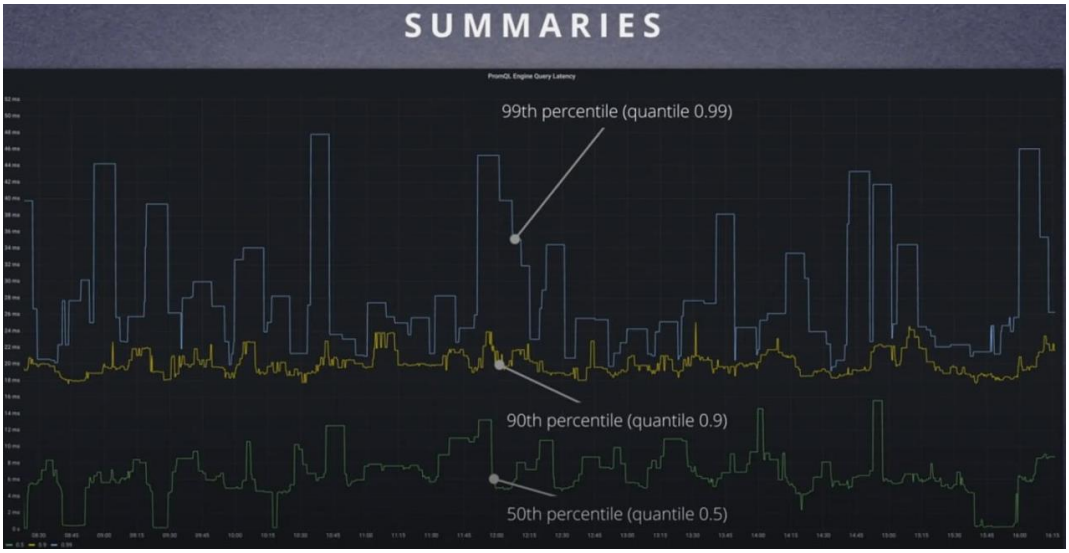


Рисунок 3 – Пример работы суммы в Prometheus

Можно заметить, что это все еще похоже на первый тип, однако мы определяем среднее за определенный промежуток. Полезно чтобы узнать распределение задержек запросов.

На графике можно заметить перцентили. Это определенный параметр, который дает представление о текущей ситуации. Представим, что у нас много запросов для приложения. Так как оно может взаимодействовать с разными другими сервисами, то это будет сетевым взаимодействием. Но сети зачастую не самый надежный способ передачи информации. Вполне возможно, что возникнет аномалия, к примеру на другой стороне наш запрос потерялся, либо к нам не пришел ответ. Из-за этого пользователь чей запрос обычно обрабатывался за секунду мог прождать десять секунд, а то и больше. Перцентиль в девяносто процентов равный пятьдесят мс означает, что из всех запросов девяносто смогли пройти быстрее чем пятьдесят мс. С одной стороны, сама по себе это не прямой показатель, что все плохо или супер хорошо, но обычно используются несколько таких же с разным значением. К примеру, 50, 90, 95 перцентилей. Было бы здорово иметь малую отдачу на 99 и 99.9 значениях, однако с повышением параметра содержать сеть стабильной становится все сложнее и сложнее.

Из этого следует, что мы получаем косвенный показатель производительности работы приложения, на основе которого можем делать выводы стоит ли заниматься оптимизацией приложения, или текущая производительность труда сопоставима с нашими ожиданиями обработки запросов. Это очень полезный тип данных в Prometheus.

Последнее что мы можем использовать это Histograms (Гистограммы), приведено на рисунке 4.

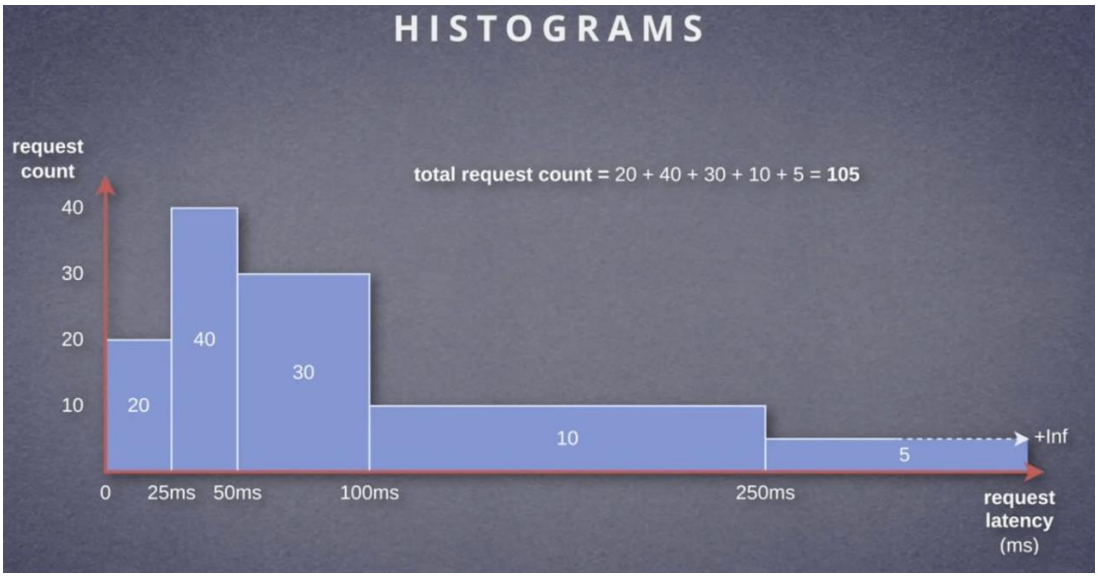


Рисунок 4 – Пример работы гистограммы в Prometheus

Это похоже на суммы, однако теперь мы можем видеть распределение в более четком виде. Сколько запросов и в каких промежутках. Стоит отметить, что на самом деле есть второй вид этой диаграммы, где мы учитываем прошлые значения. То есть было бы 20, 60, 90, и так далее. Это как раз и были процентили рассмотренные выше.

Существуют отдельные аналоги для метрик, к примеру, Victoria metrics. По своей сути она ставит такие же задачи, однако из-за простоты использования и настройки первого решения был выбран именно он.

Исходя из рассмотренных фактов мы получаем готовое решение способное обрабатывать метрики приложения, помогать нам реагировать на них и замерять производительность. Таким образом Prometheus является стандартным решением в крупных компаниях.

1.6.4 Логирование

Вдохновившись идеей центрального хранилища для метрик, люди задумались о создании подобной реализации для сбора логов. Из предложенных вариантов графиков данных для Prometheus, нельзя однозначно узнать какие были события внутри запроса. Пользователи часто не являются одним человеком, у которого есть доступ к системе. Из этого возможна ситуация, когда обрабатывается несколько обращений в один момент времени. Для дополнительной информации мы можем выводить логи. Обычно они появляются в выводе программы, но в среде Docker существует отдельное место, где файлы представляют собой это хранилище.

Loki [6] подобно Prometheus хранит логи в виде временных последовательностей. Разработчики решили, что они не будут индексировать все данные. Вместо этого индексируется только мета информация в виде labels.

Это позволит выделить цепочку сообщений, исходящих из одного источника.

Сама система имеет плюсы для нас:

- Для выгрузки логов существует свой язык запросов LogQL. Подобно PromQL он достаточно гибкий, чтобы справиться со своей задачей;
- После настройки для сбора есть доступ к визуализации данных с помощью Grafana. Как вариант можно создать доску с метриками и логами в одном месте;
- Есть разделение на прием хранение информации, а также запросов. Это позволяет масштабировать систему горизонтально при возрастающей нагрузке;
- Проста в использовании. Внедряя агента Promail[7] мы обеспечиваем автоматическую доставку логов из контейнеров в Loki.

Как выглядит путь разработчика с этими инструментами?

Мы получаем уведомление, что в метриках пользователи не могут обратиться по нужному адресу. Смотрим в метрики и убеждаемся, что это явление не случайный всплеск и не исправляется.

Делаем запросы к системе логирования, получаем информацию, где обрывается работа приложения. Находим индивидуальный идентификатор, если необходимо и отслеживаем путь запроса. Понимаем, где мы сломались и готовим исправленную версию.

На основе этой теоретической части можно приступить к созданию конечной системы.

2 РАЗРАБОТКА ПРОГРАММНОГО РЕШЕНИЯ

2.1 Архитектура проекта

Для реализации поставленных задач будет использована следующая схема, представленная на рисунке 5.

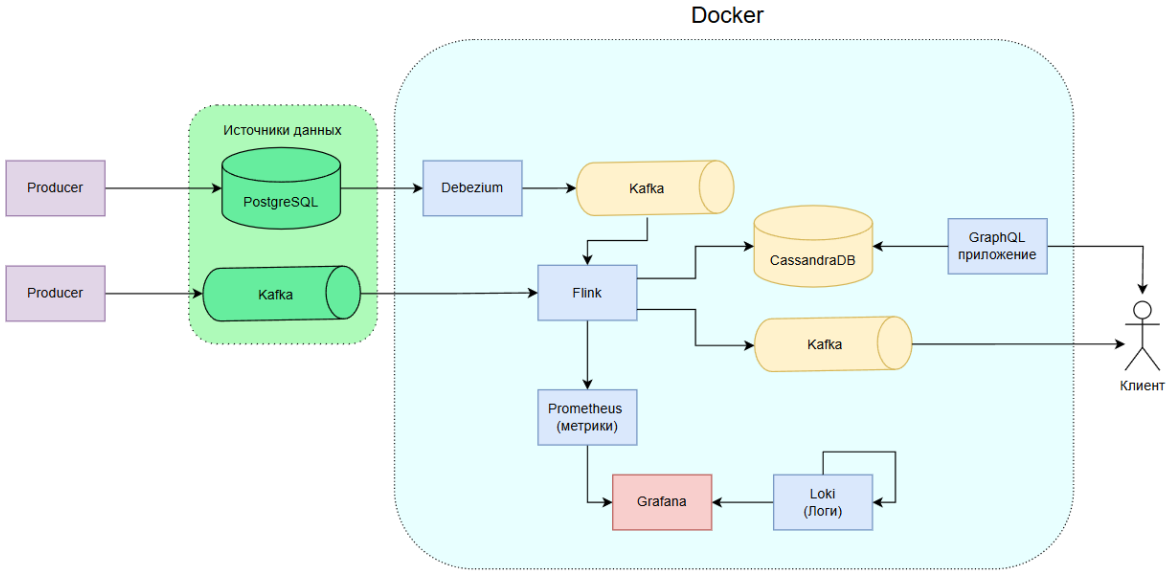


Рисунок 5 – Архитектура системы

Программный код состоит из следующих компонентов: producer сообщений, PostgreSQL, Kafka[8], Debezium[9], Flink, CassandraDB, Prometheus, Loki, Grafana[10], GrahpQL приложение.

Подробнее рассмотрим акторов системы и стек технологий.

2.1.1 Producer

Для эмуляции работы системы создан отдельный источник новых данных, так как необходимо замерить возможности системы. Он использует фреймворк Java Spring [11].

Функции Producer:

- Запись новых сообщений в топики Kafka напрямую. Доступны следующие возможности. Запись сообщения, содержащего все данные (полная доступная информация), запись сообщения содержащего id пользователя, а также перечислены поля, которые изменились;

- Изменение в базе данных. Приложение имеет доступ к базе данных, способно создать новые записи, а также изменять текущие поля в имеющихся записях;
- Быстрая работа приложения для создания нагрузки в основном контуре.

Стоит отметить, что используется он для насыщения информацией хранилищ информации, чтобы симулировать работу микросервисов, которые имеют доступ до хранения.

2.1.2 Хранение источников данных

Так как описано выше, необходима сущность PostgreSQL, которая будет хранить текущие значения для пользователей.

В ней создается отдельная таблица, в которую и поступают изменения. Сама база не посылает напрямую информацию об изменениях, поэтому этим будет заниматься Debezium.

В условиях реальной задачи она может хранить последние актуальные состояния для изначальных пользователей и помогать им получать соответствующую им информацию. Стоит отметить, что обычно она используется вместе со своей репликой, однако требует синхронного ожидания изменений, если мы хотим обеспечить надежность полученного решения. Пример как можно представить несколько таблиц представлен на рисунке 6.

Также используется Kafka кластер с использованием Zookeeper.
Она служит для хранения событий, поступающих в ее топики. В реальном проекте открываются права на доступ к конкретным сохраненным событиям в Kafka. После чего она готова к работе. Также для обеспечения надежности можно задавать количество реплицированных партиций в текущий момент, что позволит на лету заменять трафик на использование ими, если основная машина, на которой находится патриция откажет. В рамках выпускной квалификационной работы брокеры находятся внутри Docker окружения, однако в больших компаниях может быть разделение для повышения

отказоустойчивости. Стоит заметить, что есть возможность увеличить размер сообщения, если не будет хватать его для текущих задач, либо разбить одно большое на несколько отдельных.

Функционал Kafka

- хранить необработанную информацию для читателей топиков,
- мгновенная работа, так как сама Kafka понимает информацию как набор битов, из-за чего простые операции с ними происходит быстро,
- поддержание собственных копий, для сбоев отдельных машин.

Как итог финальная система хранения событий может представлять собой схему из топиков представленных на рисунке 7.

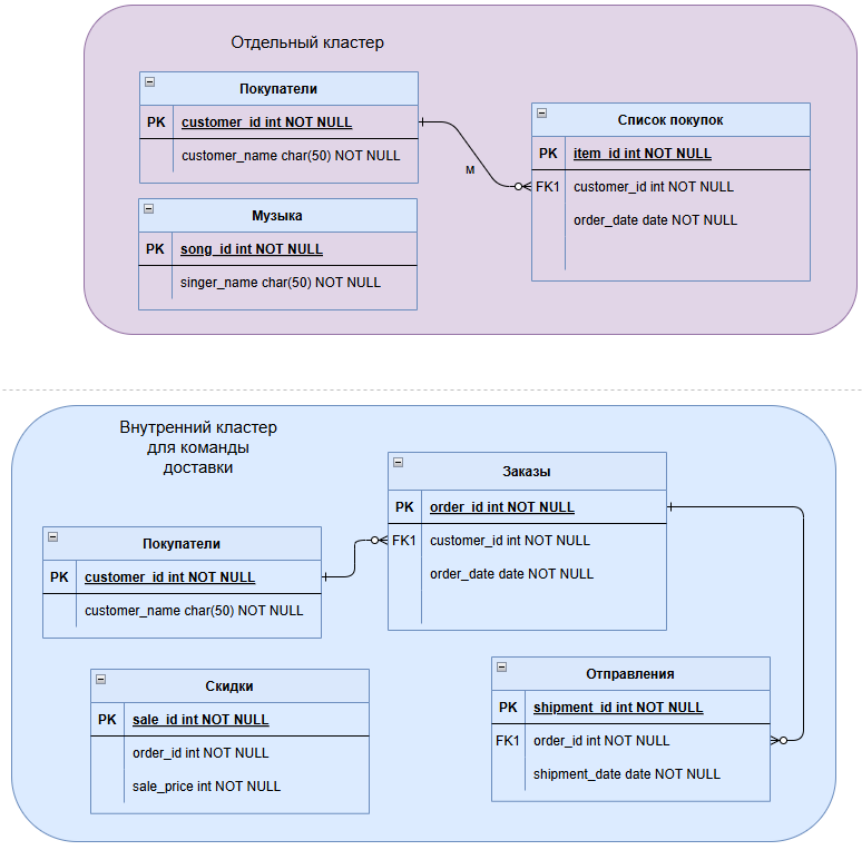


Рисунок 6 – Схема расположения распределенных баз данных

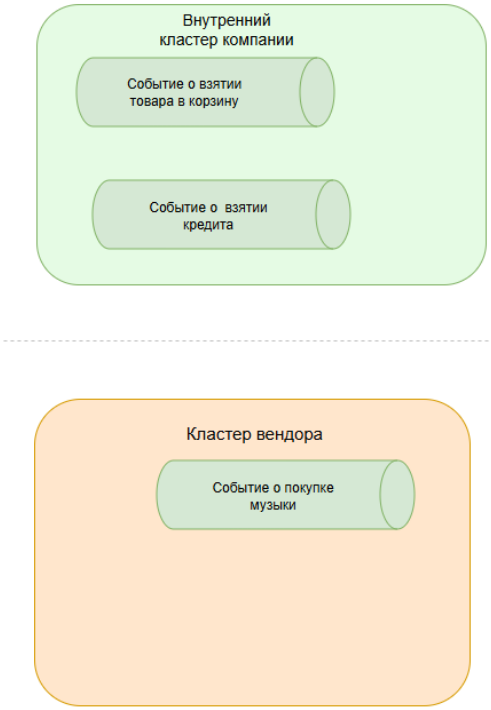


Рисунок 7 – Схема расположения кластеров Kafka

2.1.3 Debezium

Использует захват изменений данных (Change Data Capture) из баз данных и последующую передачу их в Kafka в виде событий. Для работы и отлавливания такого поведения прописывается конфигурация коннектора, где мы можем указать конкретную таблицу откуда будет собираться информация. Дальше сам Debezium берет на себя роль выслеживания изменённых данных.

В функционал входит:

- Превращение изменений базы данных в события. В них входят такие операции, как: insert, update, delete. В результате получаем сообщение с перечисленными изменениями.
- Не ограниченность одной базой данных. Если мы хотим собирать информацию из разных источников, к примеру MongoDB, то будет достаточно прописать новый коннектор, как новое место, за которым будем следить. Это позволяет быстро адаптироваться в случае подключения нового проекта, либо перехода на новый стек;

- Работа в реальном времени. Не нужно производить операцию остановки времени, чтобы понять изменения. Ведется работа с журналами транзакций в базе данных, в результате на этом месте сокращаем задержки, а также не сильно нагружаем откуда собираем данные.

Более того, из-за доступа к журналу транзакций Debezium позволяет гарантировать доставку сообщения ровно один раз.

Все это помогает обеспечить обработку в почти реальном времени.

Стоит отметить, что для ускорения работы можно использовать разные форматы записи сообщения в Kafka. Например Avro, однако был использован JSON для демонстрации и большей наглядности, так как подходит больше для читаемости человеком.

2.1.4 Cassandra

Это основное хранилище информации куда без проблем может обращаться аналитик, либо аналитический сервис.

В ней создается отдельная таблица (рисунок 8), однако содержащая интересующие нас значения.

Функционал Cassandra:

- сохранение обработанных данных,
- компактное хранение данных,
- возможность делать сложные аналитические запросы.

Из коробки Cassandra позволяет автоматически шардироваться. Т.е. разделять данные на отдельные сущности базы. По индексу можно попасть в нужную ноду кластера. Также доступна репликация. Если одна из сущностей перестанет отвечать, то копия сможет ее заменить.

Это позволяет расширить Cassandra, когда будет не хватать места для записей.

2.1.5 Flink

Основным ядром для обработки информации является эта часть программы.

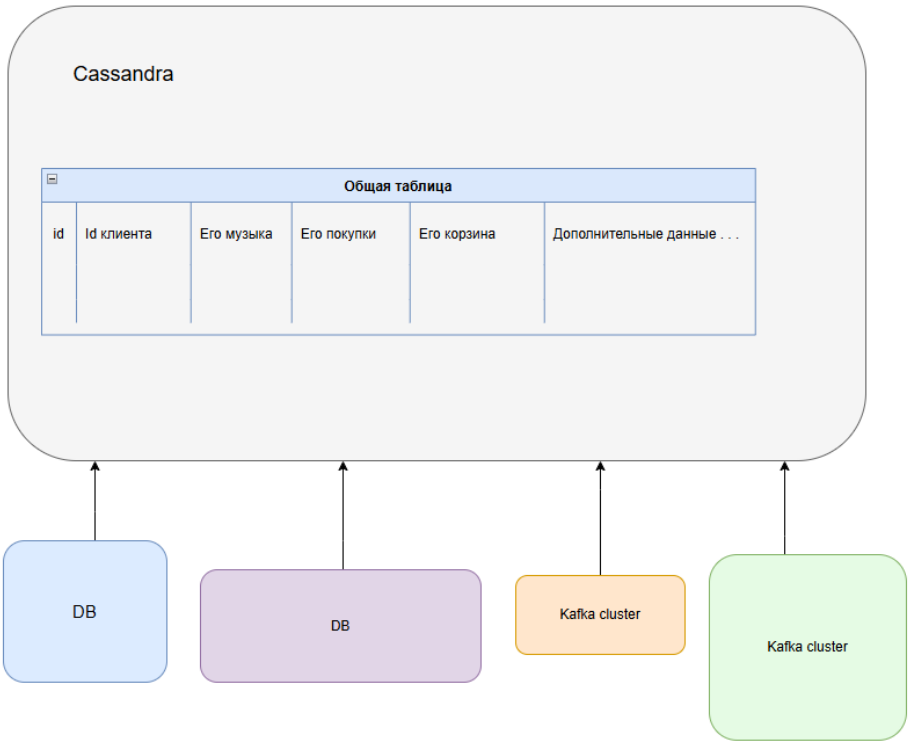


Рисунок 8 – Схемы объединения в одном хранилище профиля клиента экосистемы

Для работы разработана отдельная Job на языке Java, которая и запускается на нем.

Flink позволяет обработать данные в потоковом, а также пакетном формате. При этом обеспечивается высокая производительность и низкая задержка.

Функционал Flink:

- Доставка Exactly-once. Из-за особенностей реализации операторов и представленной теории выше, даже при сбоях можно безболезненно восстановить работу к прошлому состоянию с помощью контрольных точек;
- Ограниченные и безграничные источники. Кроме обработки в реальном времени можно, к примеру, перенести текущие данные в новую базу создав простую цепочку операторов. Что позволит порционно перейти на новый стек. Либо если мы имеем отчет в формате csv, то можно добавить его в текущую Cassandra, если мы хотим добавить специфичные данные в ручном формате сбора;

- Не ограниченность откуда и куда писать информацию. Есть множество готовых модулей работающих с стандартными стеками, позволяющее оставить возможность не сложной переделки функционала для подключения новых источников;
- Множество операторов. Так как мы не пришли к единому формату данных, то разные источники имеют свои форматы. Операторы позволяют манипулировать потоками данных, изменять их, фильтровать, соединять, распараллеливать. Это позволяет иметь гибкость к меняющимся условиям.

Рассмотрим схему работы на рисунке 9.

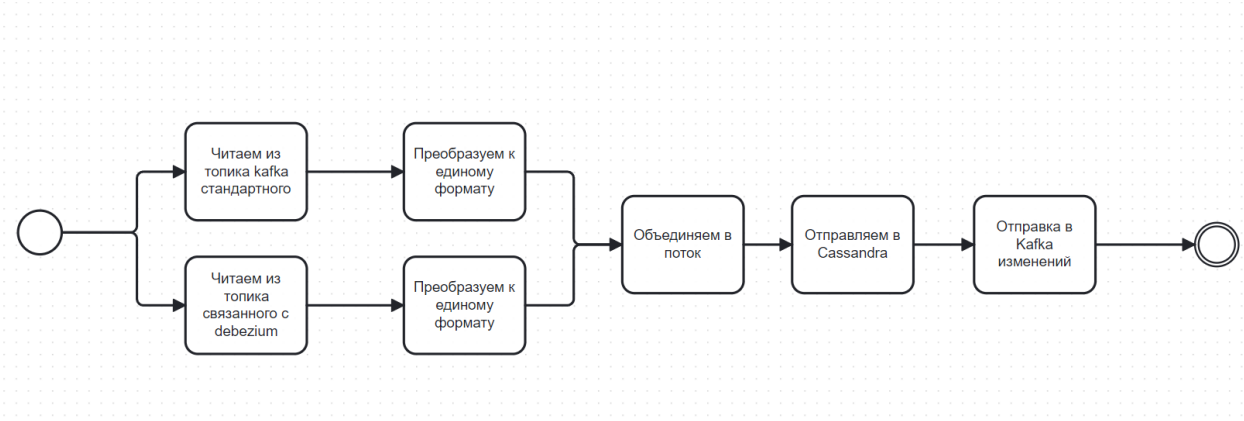


Рисунок 9 – Схема работы обработки сообщения внутри Flink

В целях демонстрации представлен основной путь работы приложения, однако в реальном проекте будет больше источников, а также больше операторов для обработки потоков.

Для повышения производительности можно включить параллелизм, в результате Flink сможет распределить нагрузку между своими узлами в кластере. Также у него уже есть интеграция с разными сборщиками метрик, а также в операторах автоматически проставлены места, чтобы их замерять. Из этого следует, что этот инструмент имеет широкую адаптивность, быструю работу и идеально подходит под наши задачи. Подробнее о том, какие еще можно использовать возможности Flink можно найти в цикле статей на Habr [12].

2.1.6 GraphQL приложение

Для взаимодействия конечного клиента (аналитика) с Cassandra служит middle слоем. После настройки схемы для запросов приложение способно передавать в ответ только запрашиваемые поля. Что снизит нагрузку на сеть.

Функционал приложения:

- Интеграция с базой данных. Фасад API позволяет скрыть от конечного пользователя взаимодействие с источником данных;
- Динамические запросы. Гибкость ответа позволяет сократить код и сделать разработку проще;
- Единый URL для запросов. Не нужно продумывать rest наименования для объектов, скрывающихся за ними.

Предполагается, что из Kafka клиент узнает, что ему не хватает и запрашивает необходимую информацию.

Функциональные части приложения развернуты в системе контейнеризации Docker [13].

3 ДЕМОНСТРАЦИЯ И ТЕСТЫ ПРОИЗВОДИТЕЛЬНОСТИ ПРОГРАММНОГО ПРОДУКТА

3.1 Сфера применения системы

Полученная системы может быть использована компаниями разных отраслей для хранения и последующей аналитики профиля клиента экосистемы компания. Можно определить его предпочтения, и в режиме реального времени подобрать наиболее ценный товар, который ему подойдет.

Особенностью решения является интеграция в уже существующие распределенные системы. Поэтому начальные ресурсы необходимые для развертывания текущего решения не очень затратны. Также возможности Flink позволяют агрегировать данные сложными фильтрами. Разработчику будет необходимо добавлять новые связи в систему, однако это подразумевает, что, сделав единожды аналитик как конечный потребитель сможет доставать необходимые данные. Debezium позволяет интегрироваться с разными базами данных и отправлять в систему хранения событий Kafka новые записи в них.

При необходимости, так как система создана из элементов, в которые их разработчиками заложена масштабируемость, можно добавить дополнительные сущности, или внутренний параллелизм для обеспечения растущих систем необходимых обслуживать.

Изначальные выбранные методы работы с источниками позволяют не нагружать их при выгрузке информации. Поэтому не нужно будет иметь санитарный день, когда выгружается информация и сделать этот процесс бесшовным, чтобы изначальный потребитель продукта не испытывал задержки при взаимодействии с фасадом системы.

3.2 Демонстрация работы системы

Изначально в Kafka поступает сообщение на рисунке 10. Оно поступает либо напрямую в kafka, либо как изменение в базе данных.

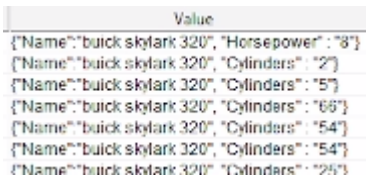


Рисунок 10 – Сообщения, поступающие в Kafka напрямую

Если рассмотрим, как это выглядит на стороне базы данных, то заметим такую картину, как на рисунке 11.

```
    "name": "postgres.public.cars.Envelope"
  },
  "payload": {
    "before": {
      "name": "dima",
      "horsepower": null,
      "cylinders": 55
    },
    "after": {
      "name": "dima",
      "horsepower": null,
      "cylinders": 55
    },
    "source": {
      "version": "1.4.2.Final",
      "connector": "postgresql",
      "name": "postgres",
      "ts_ms": 1746564348439,
      "snapshot": "false",
      "db": "exampledb",
      "schema": "public",
      "table": "cars",
      "txId": 497,
      "lsn": 23896480,
      "xmin": null
    },
    "op": "u",
    "ts_ms": 1746564348764,
    "transaction": null
```


Рисунок 11 – Сообщение в Kafka сформированное Debezium на изменение в PostgreSQL

Здесь интересует before и after, по которым можно определить, что изменилось.

После чего начинает вычитывать Flink данные из этого топика. Демонстрация как выглядит Flink операторы можно увидеть на рисунке 12.

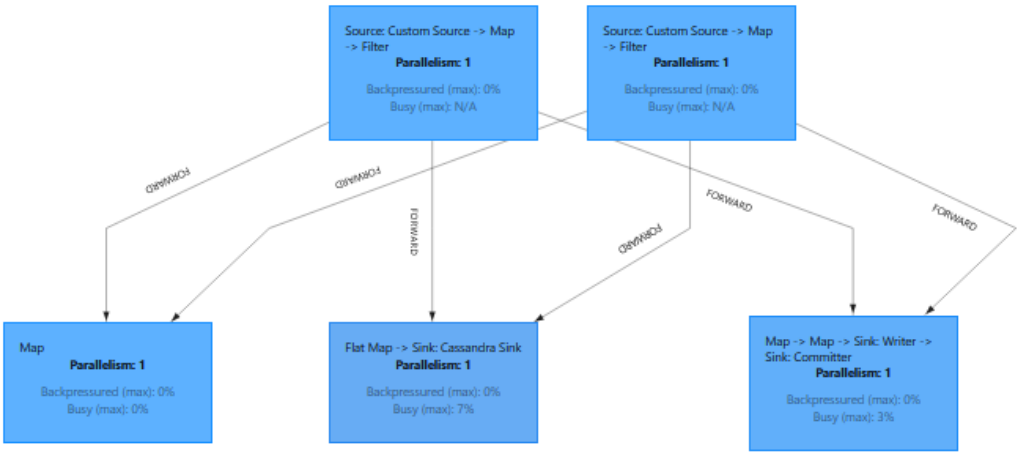
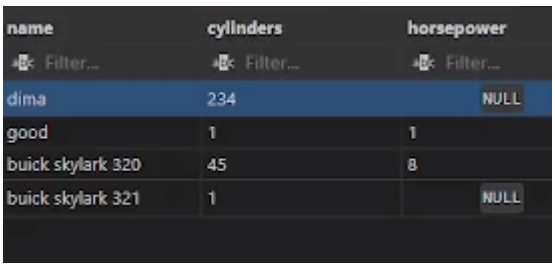


Рисунок 12 – Схема работы операторов Flink в панели Flink Job

Как итог мы получаем изменения в Cassandra. Можно убедиться на рисунке 13.



name	cylinders	horsepower
Filter...	Filter...	Filter...
dima	234	NULL
good	1	1
buick skylark 320	45	8
buick skylark 321	1	NULL

Рисунок 13 – Измененные данные в таблице Cassandra

Также будет отправлено изменение в выходящий топик Kafka. Это видно на рисунке 14.

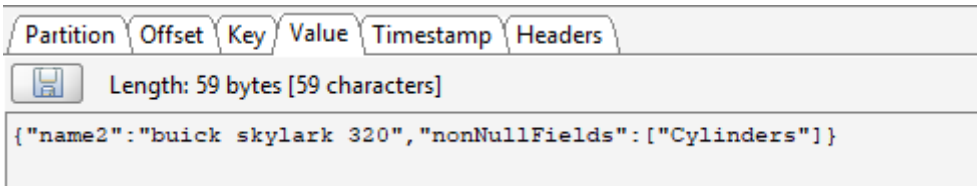


Рисунок 14 – Выходное триггер-сообщение в Kafka

Для учета логов Loki собирает данные с контейнеров о том, что происходило в конкретный момент времени. Это можно посмотреть на рисунке 15.

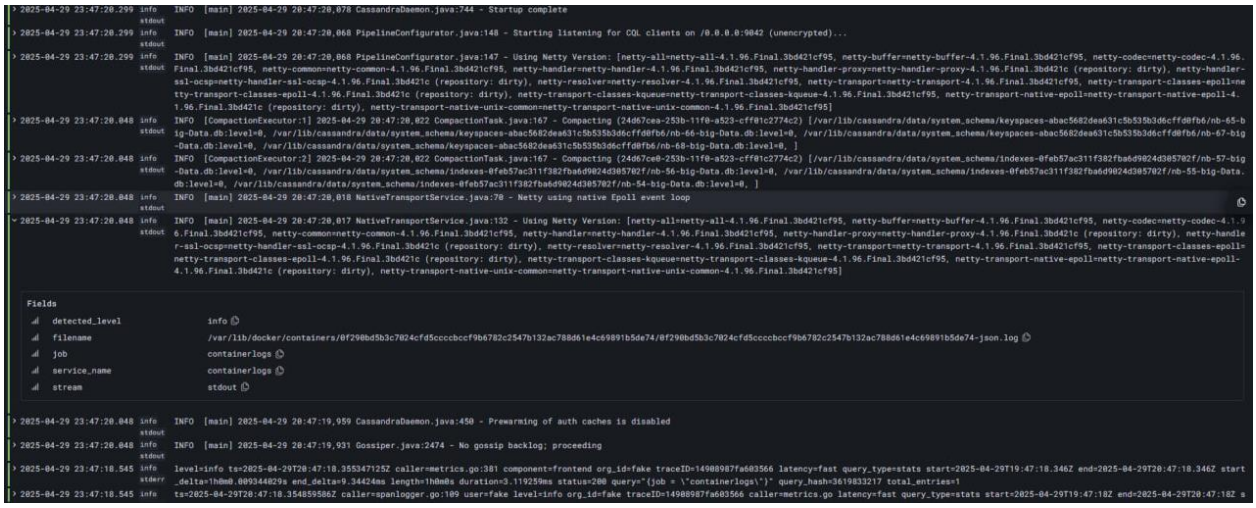


Рисунок 15 – Логи в Grafana

Метрики приложения хранятся в Prometheus. У приложения есть отдельный URL для сбора этих данных. Это показано на рисунке 16.

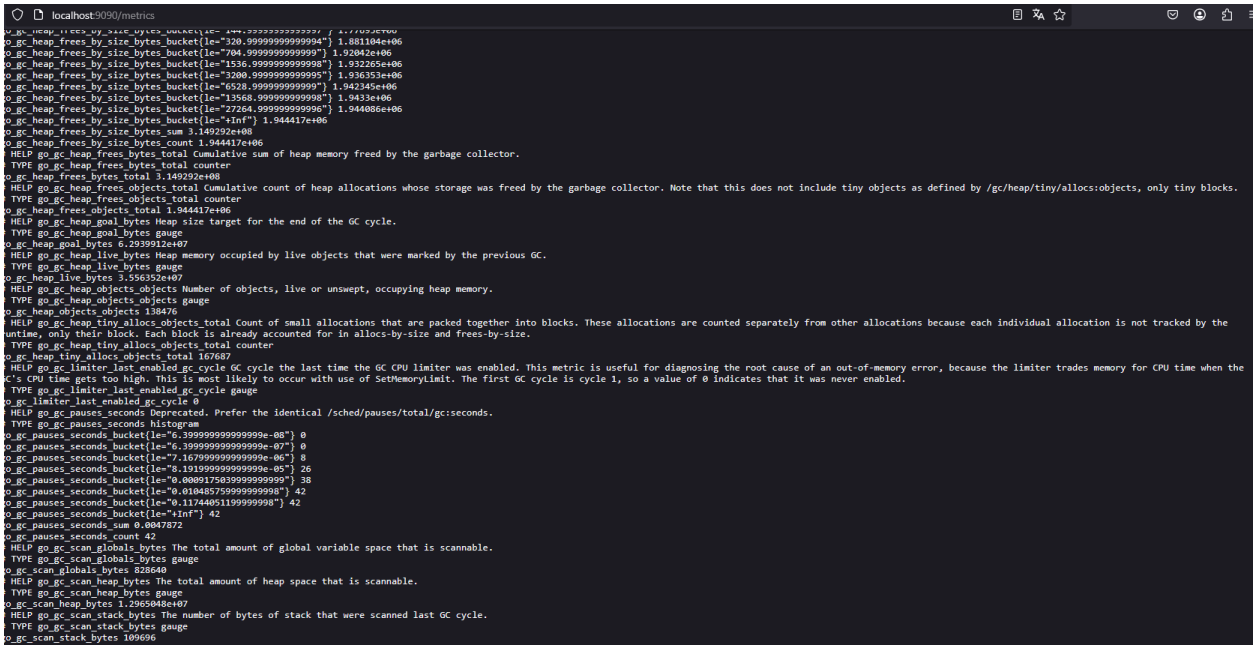


Рисунок 16 – Метрики приложения

3.3 Метрики приложения

Для замера Producer отправляет отдельные изменения, которые в итоге попадают в топики Kafka.

Замер на миллионе входных данных, на рисунке 17.

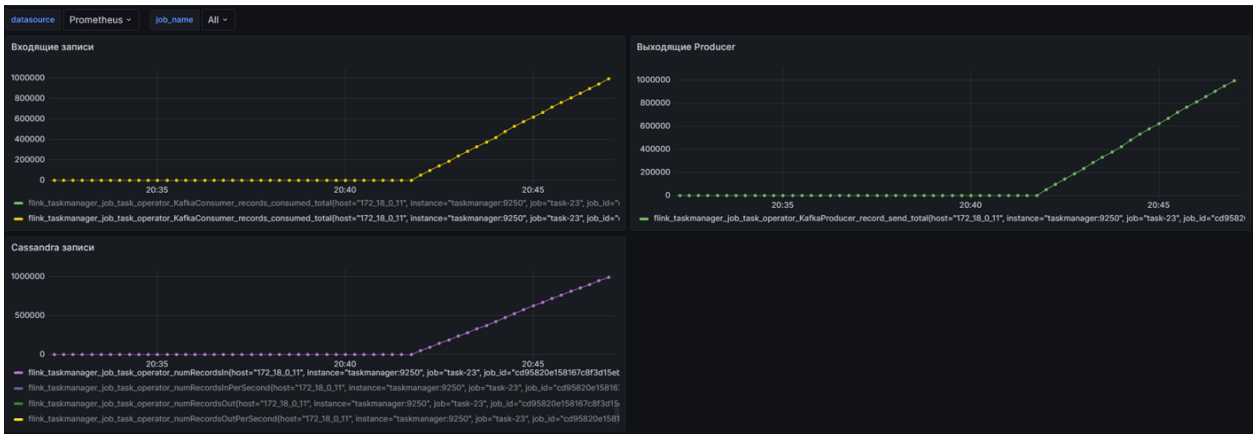


Рисунок 17 – Замер с постоянной нагрузкой на систему

Однако здесь видна линейная зависимость, так как отправитель с постоянной скоростью способен отправлять сообщения.

Но это не лучший замер системы. Для замера перезапускаю Flink Job, чтобы вычитать заново этот миллион сообщений. Flink способен поддерживать нужное смещение, чтобы этого не делать, однако для замера выбираю такую стратегию.

Рассмотрим рисунок 18 с новыми замерами.

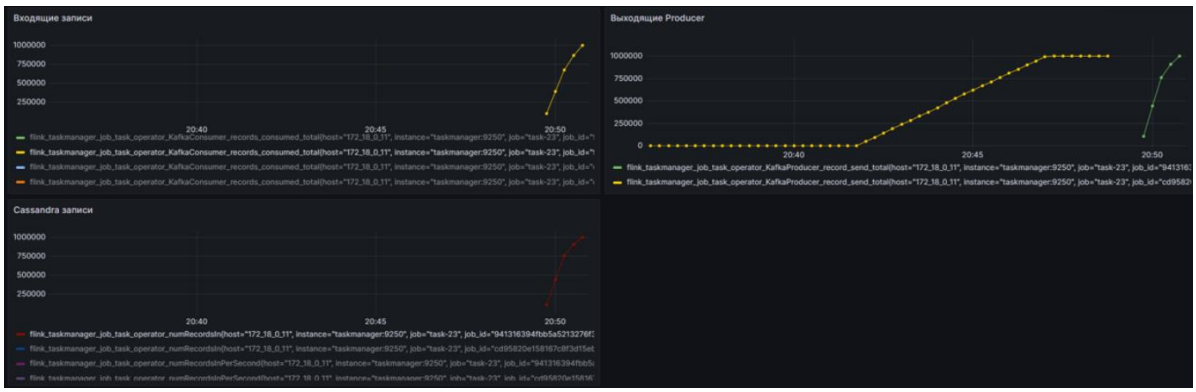


Рисунок 18 – Замер метрик с пред заполненным топиком Kafka

На нем видно, что значительно быстрее получилось обработать данные, чем в первый раз.

Так как не видно время начала и конца, то можно посмотреть на следующий рисунок 19.

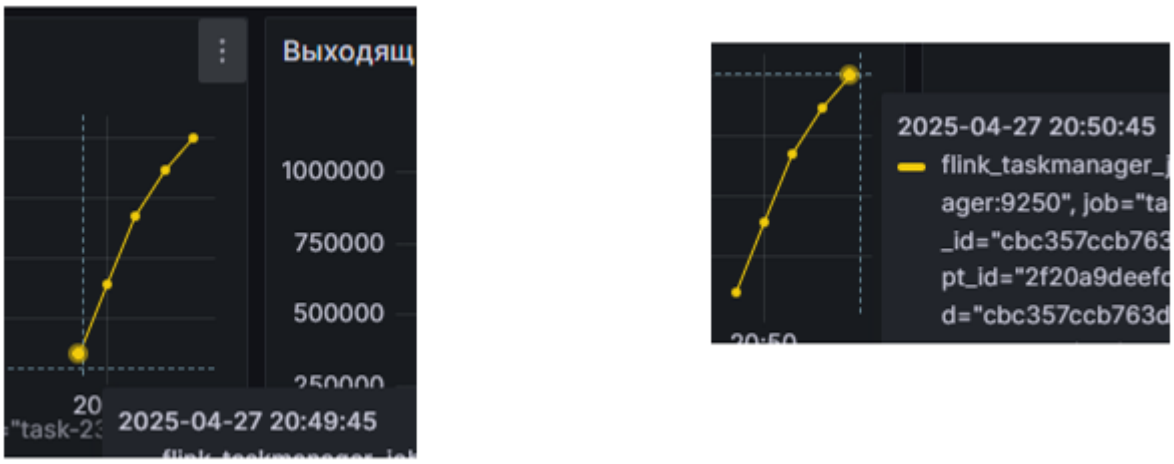


Рисунок 19 – Старт и конец замера нагрузки

На нем можно увидеть, что примерно за минуту мы смогли пройтись по миллиону поступивших мгновенно данным.

Это примерно двадцать тысяч событий в секунду. Стоит отметить, что в Flink доступны также параметры масштабирования, что означает, это не самое лучшее время за которое он может обработать все данные, но этого уже вполне достаточно для малой, средней компании.

В самом Flink можно убедиться, что мы доставили все сообщения и это заняло 40 мегабайт информации, на рисунке 20.

Bytes Received	Records Received	Bytes Sent	Records Sent	Parallelism
0 B	0	80.1 MB	1,000,110	1
0 B	0	0 B	0	1
40.1 MB	1,000,110	0 B	0	1
40.1 MB	1,000,110	0 B	0	1

Рисунок 20 – Информация об обработанных данных в панели Flink Job

Осталась самая интересная часть, замер внутренних операторов. Будем считать, что сообщения из Kafka читаются, как только так сразу, и сообщения в Cassandra поступают по возможности.

Итого мы окажемся перед гистограммой на рисунке 21.

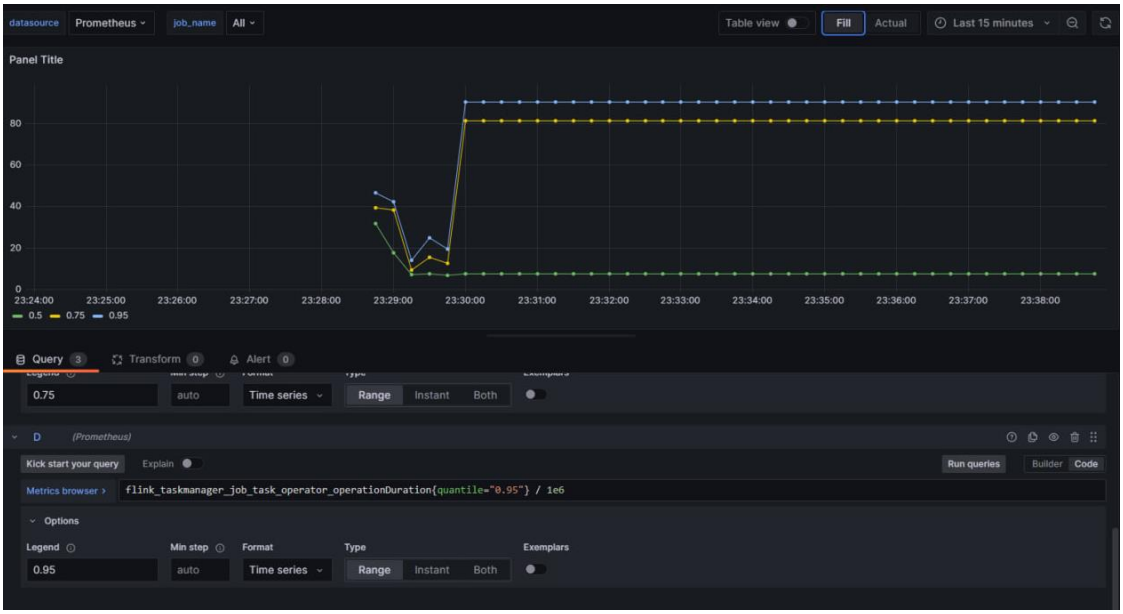


Рисунок 21 – Гистограмма времени обработки операторов в Flink

Как можно заметить если посмотреть на квантили, то увидим, что 80 миллисекунд занимает самая долгая запись от начала до конца. Но нижняя линия означает квантиль в пятьдесят процентов, то есть половина запросов быстрее чем это значение. Это 8 миллисекунд. Вполне вероятно, что первое более большое значение выходит из backpressure, так как внутри несколько операторов для преобразования и объединения, поэтому данные накапливаются внутри.

Из всех этих замеров можно убедиться, что приложения отрабатывает значительные куски информации за очень короткий период времени. Что позволит иметь наиболее актуальные данные для аналитиков, не нагружать источники.

ЗАКЛЮЧЕНИЕ

Результаты ВКР:

- разработана и интегрирована на практике инфраструктура для обработки данных в real-time и near real-time режиме,
- инфраструктура позволяет достичь высокую степень надежности и доступности, есть масштабируемость применяемых решений.

Система решает поставленную задачу в полном объеме.

Позволяет сократить время на сбор аналитических данных, а также упросить внедрение сбора новых, не значительно нагружает исходные источники.

Дальнейшее развитие системы включает в себя добавление новых источников информации, изменение системы под нужды аналитиков. Оптимизацию работы узлов кластера.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Apache Software Foundation. Apache Flink Documentation: Stable Release [Электронный ресурс]. – URL: https://nightlies.apache.org/flink/flink-docs-stable/ (дата обращения: 20.12.2024).
2. Java Documentation: 21 Version [Электронный ресурс]. – URL: https://docs.oracle.com/en/java/ (дата обращения: 24.12.2024).
3. Клеппман, М. Высоконагруженные приложения. Программирование, масштабирование, поддержка / М. Клеппман. — Санкт-Петербург: Питер, 2018. — 640 с.: ил. — (Серия «Бестселлеры O’Reilly»).
4. Уэске, Ф. Потоковая обработка данных с Apache Flink / Ф. Уэске, В. Калари; пер. с англ. В. Яценков. — Москва: Издательство, 2023. — 350 с.
5. Prometheus Documentation: 3.3.0 Version [Электронный ресурс]. – URL: https://prometheus.io/docs/introduction/overview/ (дата обращения: 27.12.2024).
6. Grafana Labs. Grafana Loki Documentation: 3.5 Version [Электронный ресурс]. – URL: https://grafana.com/docs/loki/latest/ (дата обращения: 22.12.2024).
7. Grafana Labs. Grafana Promtail Documentation: Latest Version [Электронный ресурс]. – URL: https://grafana.com/docs/loki/latest/send-data/promtail/ (дата обращения: 22.12.2024).
8. Apache Kafka Documentation: 4.0 Version [Электронный ресурс]. – URL: https://kafka.apache.org/documentation/ (дата обращения: 22.12.2024).
9. Debezium Documentation: 3.1 Version [Электронный ресурс]. – URL: https://kafka.apache.org/documentation/ (дата обращения: 27.12.2024).
10. Grafana Labs. Grafana Documentation: Latest Version [Электронный ресурс]. – URL: https://grafana.com/docs/grafana/latest/ (дата обращения: 23.12.2024).
11. VMware, Inc. Spring Framework Documentation [Электронный ресурс]. – URL: https://docs.spring.io/springframework/reference/index.html (дата обращения: 23.12.2024).

12. Ru-MTS. Цикл статей про Apache Flink [Электронный ресурс]. – URL: https://habr.com/ru/companies/ru_mts/articles/772898/ (дата обращения: 15.12.2024).

13. Docker Documentation [Электронный ресурс]. – URL: https://docs.docker.com/ (дата обращения: 23.12.2024).

ПРИЛОЖЕНИЕ А
Исходный код

На рисунке А.1 изображен QR-код со ссылкой на GitHub репозиторий с исходным кодом разработанного программного продукта.

Рисунок А.1 – QR-код на репозиторий