Санкт–Петербургский государственный университет

АБДУЛЛИН Дамир Назирович

Выпускная квалификационная работа

Разработка высоконагруженного сервиса для обработки событий и генерации реакций в приложении такси с микросервисной архитектурой

Уровень образования: бакалавриат
Направление 02.03.02 «Фундаментальная информатика и информационные технологии»
Основная образовательная программа СВ.5003.2016 «Программирование и информационные технологии»

Научный руководитель:
кандидат физ.-мат. наук, доцент
кафедры компьютерного моделирования и
многопроцессорных систем:
Корхов Владимир Владиславович

Рецензент:
программист, Общество с ограниченной
ответственностью «ТСистемс СиАйЭс»
Васюнин Дмитрий Алексеевич

Санкт-Петербург
2020 г.

Содержание

Введение

В связи с ростом темпов жизни среднестатистического человека и дороговизной содержания собственного автомобиля возрастает спрос к услугам такси. С развитием технологий пришли удобные мобильные приложения, агрегирующие множество компаний, предоставляющих услуги развозки пассажиров (таксопарков), и предоставляющие удобный интерфейс. В разработке и поддержке такого приложения возникает много проблем, которые требуют оперативного решения. Одна из таких проблем - мотивация водителей принимать заказы, которые, возможно, им не выгодно исполнять по некоторым причинам (точка А далеко от текущего местоположения кандидата на заказ). Очевидно, что случаи, когда пассажир не может уехать из-за того, что все водители отказываются от поездки, плохо влияет на их лояльность по отношению к сервису. Возникает задача мотивации водителей принимать такие заказы. На начальном этапе это происходило при помощи нескольких правил, внедренных в программный код (например, за три отказа от заказа подряд снять 1 рейтинг), которые абсолютно не покрывали требования бизнеса.

Постановка задачи

Целью работы является разработать такой инструмент для сервиса такси, с помощью которого можно генерировать реакции на события с определенными условиями. События в сервисе такси имеют разную структуру и подразумевают разные состояние заказа. Например, событие reject_manual означает отказ водителя от заказа, a order_complete - успешно выполненный заказ. Под реакцией подразумевается любое действие из заранее заданного множества (отправка push-уведомлений, начисление рейтинга, блокировка). Правила должны быть гибкими и легко конфигурируемыми, чтобы любое их изменение не привлекало к работе инженера программного обеспечения. Пример правила может быть таким: отправить push-уведомление водителю, если за последний час он отказался от трех заказов, расчетная длительность которых не превышала 20 минут. Необходимо создать новый функционал по генерации реакций на определенные события в отдельном микросервисе с удобной и масштабируемой архитектурой. Добавление новой реакции (поддержка начисления другой характеристики водителя за определенные действия) должно занимать не больше двух дней, в зависимости от ее специфики. Так, к примеру, начисление абстрактной величины будет легко поддержать в силу того, что функционал похож на уже имеющееся начисление рейтинга, а вот блокировка водителя будет требовать написания нового модуля. Таким образом приложение должно иметь модульную архитектуру и быть легко имплементируемым.

Реализовать возможность начисления рейтинга на основе заранее заданного признакового пространства с использованием алгоритма машинного обучения с целью избавиться от необходимости вручную задавать правила, по которым он начисляется, и сделать эту процедуру более гибкой.

Обзор литературы

В силу предполагаемой высокой нагрузки на разрабатываемый сервис обработку событий планируется распределить на несколько потоков. В источнике [2] автор определяет основную проблему параллельной работы программы. Он указывает на условия Бернстайна, благодаря которым можно определить, является ли последовательность команд (или их часть) детерминированной. Псевдопараллельное выполнение набора программ заключается в том, что отдельные процессы внутри себя выполняют инструкции строго последовательно, но при этом между собой управление переходит недетерминированно. Детерминированной называется такая последовательность действий, которая при любом псведопараллельном выполнении отдает один и тот же состав выходных данных. Недетерминированная последовательность действий, которая в ходе выполнения при различных псевдопараллельных выполнениях отдает различный состав выходных данных, называется также критическим местом в приложении. На основе данных определений можно выявить такие места в сервисе и решить каждую проблему по отдельности или обобщить решение на все случаи. Автор перечисляет два подхода для решения - взаимосинхронизация и взаимоисключение. Первый подразумевает отсутствие (предотвращение) пересечения множества входных данных алгоритмов, а второй - синхронизацию и блокировку данных, находящихся в общей памяти. Решение проблемы состояния гонки не должно базироваться на относительном времени выполнения процесса или количества процессоров, потоков (например, основная обработка потока входных данных длится 300 мс, поэтому новые данные отправляем каждые 400 мс). Минус такого подхода, очевидно, в том, что сложно отследить увеличение времени обработки и вовремя подкорректировать связанные с этим показатели. Впоследствии найти ошибку связанную с race condition является нетривиальной задачей.

В [10] автор предоставляет обширное множество советов и рекомендаций в разработке распределенного приложения. В книге описаны способы сохранения порядка сообщений в очереди, журналирование, репликация распределенных баз данных, особенности транзакций. В источнике [3] описаны несколько популярных и широко используемых паттернов проектирова-

ния программного обеспечения. Один из описанных там паттернов - шаблон проектирования “Канал событий”, благодаря которому реализована система подписчиков на канал событий (отправление push-уведомлений с помощью API). Паттерн Класс-строитель позволяет реализовать интерфейс действия, которое будет произведено при некоторых, заранее заданных, правилах. Он позволит в зависимости от сработавшего правила легко возвращать необходимый экземпляр класса, что позволит легко добавлять новый функционал. Способ реализации реакций основан на шаблоне Команда. Паттерн команда описывает реализацию интерфейса реакции с описанным необходимым функционалом, от которого наследуются любые действия с имплементацией этих методов.

В статье [9] произведен анализ производительности и точности алгоритмов, основанных на градиентном бустинге: Adaboost, LightGBM, XGBoost и Catboost. Авторы отмечают, что наибольшее качество и наименьшие требования к CPU предоставляет алгоритм машинного обучения, реализованный в библиотеке Catboost.

Глава 1. Проектирование сервиса

1.1 Архитектура сервиса

Начать проектирование стоит с политики взаимодействия с базой данных, в которой будут храниться обработанные и необработанные события.

1.1.1 Двухзвенная архитектура

В основе микросервисной архитектуры лежит общение по принципу клиент-сервер (например, по протоколу HTTP с внутренней аутентификацией). Двухзвенная архитектура используется в приложениях, где сервер отвечает на запросы самостоятельно, без использования сторонних источников или сетевых приложений. Схематически архитектура представлена на рис. 1.

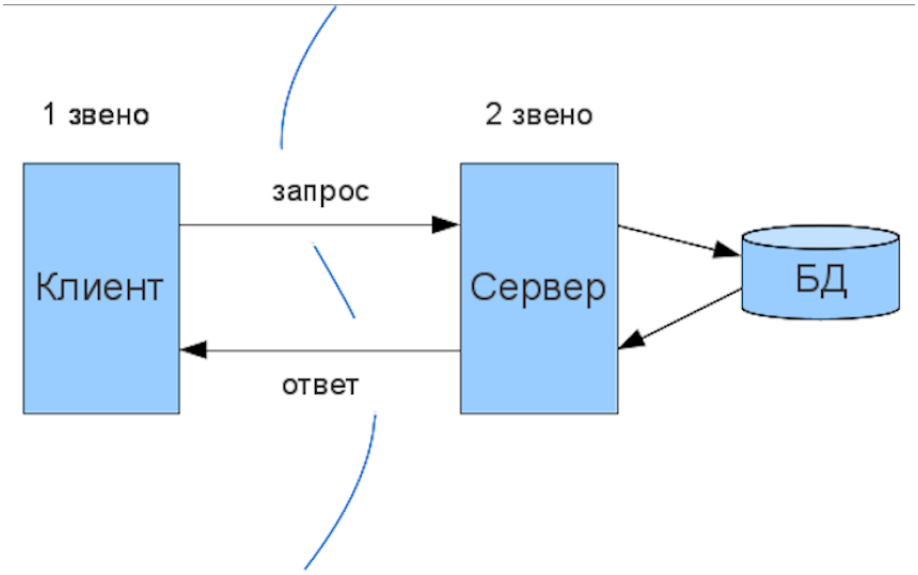

Рис. 1: Двухзвенная архитектура.

Рассмотрим 2 варианта реализации двухзвенной архитектуры:

- Сервер БД
- Сервер приложений

Сервер БД

Рассмотрим требования, представляемые по отношению в хранилищу данных, которые сервер базы данных гарантирует:

- В любой момент времени данные должны быть непротиворечивы. Это требование заключается в соблюдении всех типов данных, ограничениях, наложенных на содержимое. Например, дата создания события представляется в виде неотрицательного целого числа (timestamp), но ограничений сверху не имеется, потому что мы можем запланировать создание события, если хотим, чтобы оно обработалось позже

- БД должна отражать некоторые правила предметной области, законы, по которым она функционирует (business rules). Количество событий вида заказ могут отражать “здоровье” компании в целом, ведь отражают основную продуктовую метрику

- Необходим постоянный контроль за состоянием БД. При резком увеличении количества событий возможна необходимость подключение дополнительного узла в распределенной системе. При увеличении количества блокировок водителей, возможно стоит задуматься о смягчении правил блокировок и т.д.

Основу данной модели составляют:

- Механизм хранимых процедур как средство программирования SQL-сервера [12]

- Механизм триггеров как механизм отслеживания текущего состояния информационного хранилища

- Механизм ограничений на пользовательские типы данных, который иногда называется механизмом поддержки доменной структуры

Схематическое представление архитектуры представлено на рис. 2

Общение между сервером и клиентом происходит на некотором диалекте SQL (в нашем случае это pgSQL). На стороне сервера непосредственная

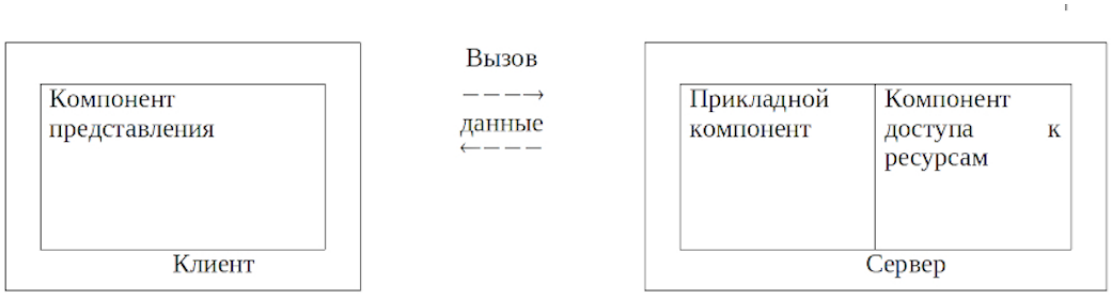

Рис. 2: Архитектура сервера БД.

работа с БД происходит за счет хранимых процедур, что позволяет сэкономить время на построении плана выполнения запроса, его оптимизации и т.п.

Данную модель поддерживают большинство современных СУБД: MySQL, PostgreSQL, Oracle. Такая архитектура позволяет избежать ошибок на стороне поставщика данных за счет системы триггеров. Но есть и существенные недостатки такие как:

- Ограниченные возможности и функционал, предоставляемые посредством реализации хранимых процедур. Их возможности гораздо беднее, чем функциональные возможности высокоуровневых языков (Python, C++, Java)
- Ограничения, предоставляемые конкретной СУБД, отсутствие тестирования и отладки хранимых процедур
- Большая нагрузка на сервер

Сервер приложений

Данное решение является логическим следствием из сервера БД. Целью является расширить функционал хранимых процедур, снизить требования к конфигурации клиента за счет повышения требований к производительности, безопасности и надежности сервера. Архитектура представлена на рис. 3. Прикладной компонент выделен как важнейший изолированный элемент приложения. Для его определения используются универсальные механизмы многозадачной операционной системы, и стандартизованы интерфейсы с двумя

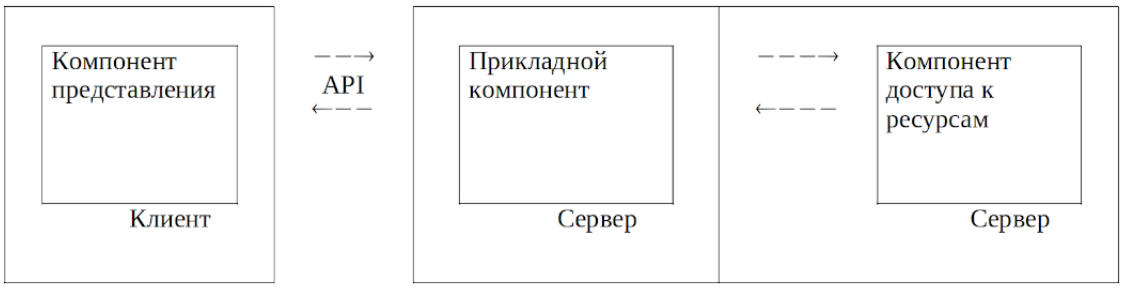

Рис. 3: Архитектура сервера приложений.

другими компонентами. Таким образом прикладной компонент может предоставить те возможности поддержания непротиворечивости данных, которые не могут быть реализованы средствами механизма триггеров.

1.1.2 Трехзвенная архитектура

Последний рассматриваемый вариант модели клиент-серверной парадигмы - трехзвенная архитектура. В силу высоконагруженности сервиса появляется необходимость в реализации распределенных вычислений. Такой подход может быть реализован по нескольким причинам:

- Несвязанность событий между собой, есть только одна необходимость - обработка событий одного водителя в одном процессе, чтобы избежать состояния гонки данных
- Предоставлены технические ресурсы и инфрастуктурная поддержка
- СУБД, позволяющие реализовать работу с распределенными вычислениями. Реализации блокировок, реплицирование и т.п.

Схематически архитектура представлена на рис. 4.

Одно из главных преимуществ трехзвенной архитектуры заключается в том, что сервер приложения может быть представлен двумя или более частями, каждая из которых может выполняться на отдельном вычислительном устройстве. Выделенные звенья общаются между собой в заранее согласованном формате.

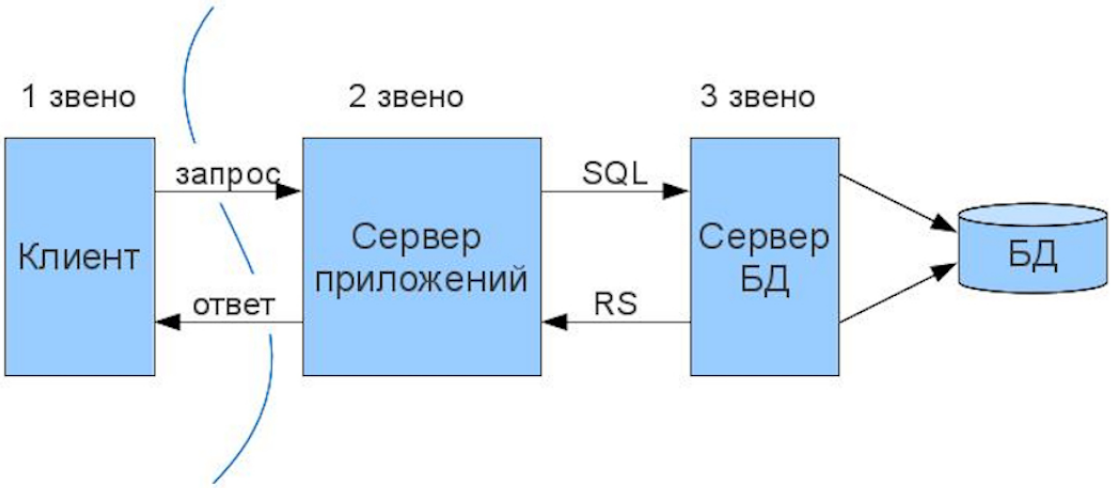

Рис. 4: Трехзвенная архитектура клиент-серверного приложения.

Как правило, третьим звеном в трехзвенной архитектуре становится сервер приложений, т.е. компоненты распределяются следующим образом:

- Представление данных — на стороне клиента
- Прикладной компонент — на выделенном сервере приложений (как вариант, выполняющем функции промежуточного ПО, гарантирующий основную согласованность данных)
- Управление ресурсами — на сервере БД, который представляет запрашиваемые данные

Двухзвенная архитектура проще в реализации и эксплуатации, так как все запросы обслуживаются одним сервером, но именно из-за этого она менее надежна и предъявляет повышенные требования к производительности сервера. Трехзвенная архитектура сложнее, но благодаря тому, что функции распределены между серверами второго и третьего уровня, эта архитектура представляет:

- Высокую степень гибкости и масштабируемости
- Высокую безопасность (т.к. защиту можно определить для каждого сервиса или уровня)

- Высокую производительность (т.к. задачи распределены между серверами)

Вышеперечисленные преимущества трехзвенной архитектуры полностью покрывают требования, поставленные перед задачей. Клиент представлен сервисом обработки событий, сервер - хранилищем событий, который представляет некоторое унифицированное API, которое покрывает необходимый функционал по представлению данных клиенту, и сервер БД.

1.2 Отказоустойчивость

Ниже представлены возможные точки отказа разрабатываемого приложения:

- Физическая недоступность кластера
- Ошибка в коде
- Некорректные входные данные
- Недоступность зависимых сервисов

Возможные пути решения для каждого из отмеченных пунктов представлены далее.

Физическая недоступность кластера

Данный вид проблемы может возникать по многим причинам, таких как физические неполадки на стороне кластера, проблемы с программным обеспечением, ошибка системного администратора и т.п. Чтобы избежать недоступности сервиса (downtime), следует распределить его на несколько несвязанных физических кластеров. Таким образом при поломке одного из серверов выполнение происходит на оставшихся работоспособных кластерах. Важно проводить периодические “учения” по имитации недоступности одного из хостов, чтобы быть уверенным в достаточности ресурсов для работы в рамках увеличенной нагрузки.

Ошибка в коде

Никто не застрахован от человеческого фактора при написании кода, но есть множество способов минимизировать ущерб от этого.

1. Покрытие unit-тестами [8]. Целью при разработке было держать покрытие кода тестами около 95 процентов, чтобы быть уверенным в результатах обработки

2. Интеграционные тесты [14]. В силу микросервисной архитектуры важно не сломать взаимодействие сервисов между собой изменением в коде и сохранять обратную совместимость API

3. Тестирование в разработческом окружении с полной имитацией цикла заказа такси

4. Включение нового функционала с фолбеком или иметь возможность быстро его выключить. Фолбек подразумевает поведение, когда в случае возникновении ошибки в новом функционале должна быть возможность использовать старый вариант или значениея по умолчанию

Некорректные входные данные

При возникновении некорректных входных данных необходимо предотвратить их сохранения в базу данных, чтобы избежать не ожидаемых реакций на них, а также настроить мониторинг на появление таких ситуаций. Важно быстро реагировать на подобные инциденты, так как они могут сигнализировать о нездоровье сервиса в целом.

Недоступность зависимых сервисов

Разработчик отвечает только за те сервисы, поддержку которых он ведет. Поэтому важно при проектировании графа зависимостей разрабатываемого приложения определить “сильные” и “слабые” связи. Сильной связью будем называть такую связь, которая может быть проигнорирована в случае недоступности связанного узла. Слабая же связь может быть опущена с деградацией функционала (например, использование значений по умолчанию). На рис.5 представлен граф для проектируемого приложения. Необходимо об-

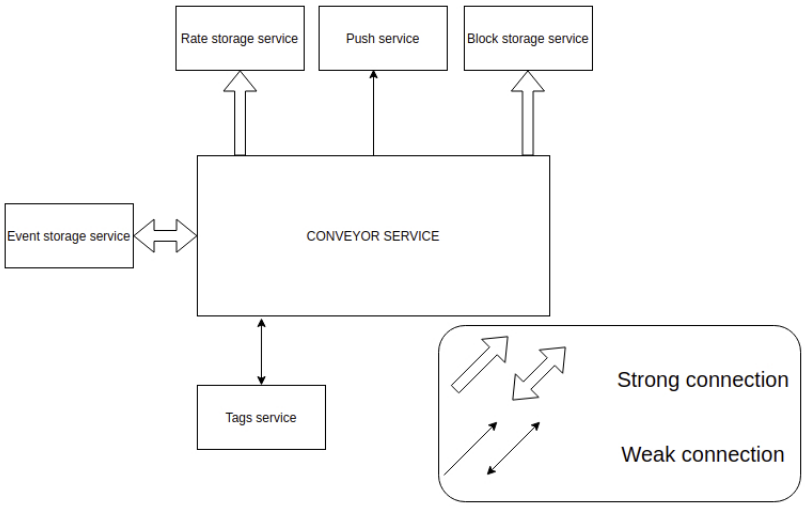

Рис. 5: Граф связей сервиса.

рывать цикл обработки сообщения при возникновении неполадок на узлах с сильной связью с дальнейшим перепланированием его выполнения. Имеет смысл использовать небольшую задержку перед следующим выполнением, чтобы увеличить вероятность “выздоровления” зависимого сервиса к моменту повторной обработки задачи. Также следует использовать добавленную к постоянной задержке случайную величину, распределенной по конкретному закону распределения, с целью избежать пиковой нагрузки на определенный момент времени.

В случае повторного выполнения одной и той же задачи (перепланирование из-за ошибки во время первой обработки) важно соблюдать идемпотентность [18] всех операций, изменяющих состояние контекста. Особенно важно соблюдать ее при вставке данных в очередь задач. Для этого можно задать идентификатор задачи, формируемый на основе метаданных задачи, который позволит однозначно определять заказ и его состояние. Например, идентификатор заказа и индекс, обозначающий количество обновлений состояния заказа.

1.3 Параллельная обработка

В силу распределенности приложения, а также обработки в несколько потоков, следует предусмотреть возможные ошибки, которые могут возникнуть на этой почве, а именно ситуации гонки.

Рассмотрим ситуацию на примере. Предположим водитель два раза подряд за небольшой промежуток времени отказался от выполнения заказа. Два сообщения отправились в очередь для обработки, но попали на разные кластеры. Получается, что в ходе выполнения кода происходит обращение к одному и тому же контексту и возможно одновременное его изменение. В нашей ситуации из стороннего источника было получено значение рейтинга 50 в обоих циклах обработки, далее на основе вычислений было получено значение, на которое нужно уменьшить рейтинг водителя за отказ от выполнения заказа, допустим на 10. Каждый инстанс приложения отправляет в источник запрос с уникальными токенами идемпотентности (потому что события фактически разные) запрос на установление нового значения рейтинга - 40. Возникает типичная ситуацию гонки состояний, которую необходимо решить, чтобы избежать большого количества обращений в службу поддержки по вопросам о некорректном начислении рейтинга.

Пусть каждому потоку обработки присвоен целочисленный уникальный идентификатор:

$$0 < i <= n, n - \text{количество потоков обработки} \tag{1}$$

При обращении к очереди за новой пачкой необработанных сообщений следует отправлять этот идентификатор и число n. При формировании запроса к СУБД используем такое условие с использованием хэш-функции [2]:

$$hash(driverId) \mod n == i \tag{2}$$

Утверждение: Пусть n - количество потоков обработки, i - уникальный идентификатор потока обработки заданный в 1, а hash - детерминированная хэш-функция, тогда условие 2 гарантирует обработку одной сущности в

рамках одного потока.

Доказательство: От обратного: предположим, что сущность с идентификатором id попало в обработку в два разных потока с идентификаторами i и j, а всего потоков n. Так как по условию для запроса необработанных сообщений используется условие 2, оно выполняется для обоих потоков, то есть:

$$hash(id) \quad mod \quad n == i \quad \text{и} \quad hash(id) \quad mod \quad n == j$$

Что противоречит условию детерминированности хэш-функции и различию потоков.

1.4 Обработка сообщений

1.4.1 Данные для обработки

Заказ формируется на стороне поставщика событий, затем передается очереди задач на планировку в обработку сервиса. Заказ представлен в формате json, значения которого являются уникальный идентификатор события, его имя и тип, зона, в которой это событие произошло, timestamp, некоторый идентификатор водителя и другие метаданные. Дополнительные необходимые поля можно получить из большого key-value хранилища [5], где представлена вся актуальная информация о заказе. Выбор именно key-value базы данных объясняется особенностью структуры и необходимостью быстрого доступа по ключу к слабоструктурированной информации об объекте.

1.4.2 Структура правил

Ранее все правила были реализованы в коде и имели примитивную структуру. Минусы такого подхода:

- Плохая масштабируемость
- Сложность редактирования (приходится писать код, проходить ревью, релизить код)

- Сложность понимания (правила формируются менеджерами, многие из которых не разбираются в программировании)

Для удобства визуализации, использования и редактирования было принято решение использовать json формат [7], который позволит решить две последние проблемы. Первый уровень будет задаваться зоной, в которой событие произошло. Например moscow, spb, urengoi и т.п.. Также зададим некоторую зону под названием default, в которой можно описать общие случаи, применимые ко всем зонам для экономии места. На следующем уровне расположится массив из правил, которые буду иметь следующую структуру:

```
{
    actions: [{
                action: {
                    name: string,
                    type: string,
                    # additional properties specified for
                        each action type
                },
                tags: Optional[tag_predicate]
        }],
    disabled: Optional[bool],
    tags: Optional[tag_predicate],
    events: [{
                name: string,
                type: string,
                events_to_trigger_count: Optional[int],
                events_to_trigger_period: Optional[int],
                tags: Optional[tag_predicate],
    }]
}
```

Напротив каждого поля указаны тип данных, который должен быть передан в качестве аргумента. Конструкция вида Optional[int] означает, что поле не является обязательным и будет задано значением по умолчанию в случае

отсутствия. Поле actions описывает действия, которые будут предприняты в случае, если все условия будут выполнены. Они заданы массивом, чтобы предоставить более гибкий инструмент. Каждое действие имеет поле tags, которое позволяет применять действия только при условии, что водитель или событие имеют определенный набор тэгов. Тэгом мы называем некоторый динамический признак сущности. Под динамическим подразумевается, что такой признак в реальном времени часто меняется: добавляется или удаляется. Хранить такие признаки как булевы поля слишком дорого и в плане занимаемого места, и в плане скорости обработки. В качестве примера можно привести тэг дальности поездки: long trip, medium trip, short trip. Применение становится очевидным - в случае выполнения заказа мы хотим по разному наградить водителя в зависимости от того, сколько времени он потратил на его выполнения. Преимущество такого подхода заключается в крайней гибкости:

- Нет необходимости добавлять изменения в коде, если со стороны бизнеса приходят новые требования
- Нет ограничения в природе тэгов
- Возможность составления сложных условий

Один из типов данных tag predicate, который представляет из себя json объект, представляющий из себя предикат. Пример:

```
{
        operation: OR,
        operands: [{
                #  another tag_predicate
        }, # other operands
        ]
}
```

И последнее поле - events. Оно отвечает за события, которые должны послужить триггером для действий. Можно указать количество необходимых событий (events_to_trigger_count), временной промежуток рассматриваемых событий (events_to_trigger_period) и, опять же, тэги.

1.5 Результаты

В результате был разработан масштабируемый микросервис, способный обрабатывать поток событий без рисков возникновений ошибок в силу параллельной обработки сущностей. Добавление новых правил для генерации реакций занимает до двух минут и без необходимости перезапускать приложение или изменять его код. Ниже в таблице показаны результаты, которые были достигнуты благодаря новому подходу.

Таблица 1: Сравнние подходов

	Старый подход	Новый подход
Скорость обработки одного события	1-2 мс	60 мс
Количество реакций (правил)	3	800
Возможные реакции	начисление рейтинга	начисление рейтинга; отправка push-уведомлений; блокировка водителей; изменение состояния водителя благодаря другим сервисам
Время добавления новой реакции	2 дня	5 минут

Тестирование производительности и отказоустойчивости приложения происходило по следующему сценарию:

1. Развертывание приложения на 3 кластера
2. Включение поставки сообщений в очередь

3. Включение обработки и проверка на корректность выполнения цикла обработки

4. Проверка нарушений связности с зависимым сервисами

    - Для узлов с сильной связностью цикл должен обрываться и перепланироваться)
    - Для узлов со слабой связностью цикл должен продолжать выполнение, используя поведение по умолчанию)

5. Выключение обработки на одном из кластеров с целью проверить отказоустойчивость приложения

6. Выключение обработки на всех кластерах, чтобы накопить очередь сообщений и оценить скорость обработки большого количества сообщений

Все пункты тестирования были успешно выполнены. Среднее время обработки сообщения - 60 мс, обработка одной сущности - 65 мс. В силу асинхронной обработки всех сообщений среднее время обработки пачки событий (10 сущностей по 1-2 событию) занимает 100 мс. Процесс обработки изображен на рис.6.

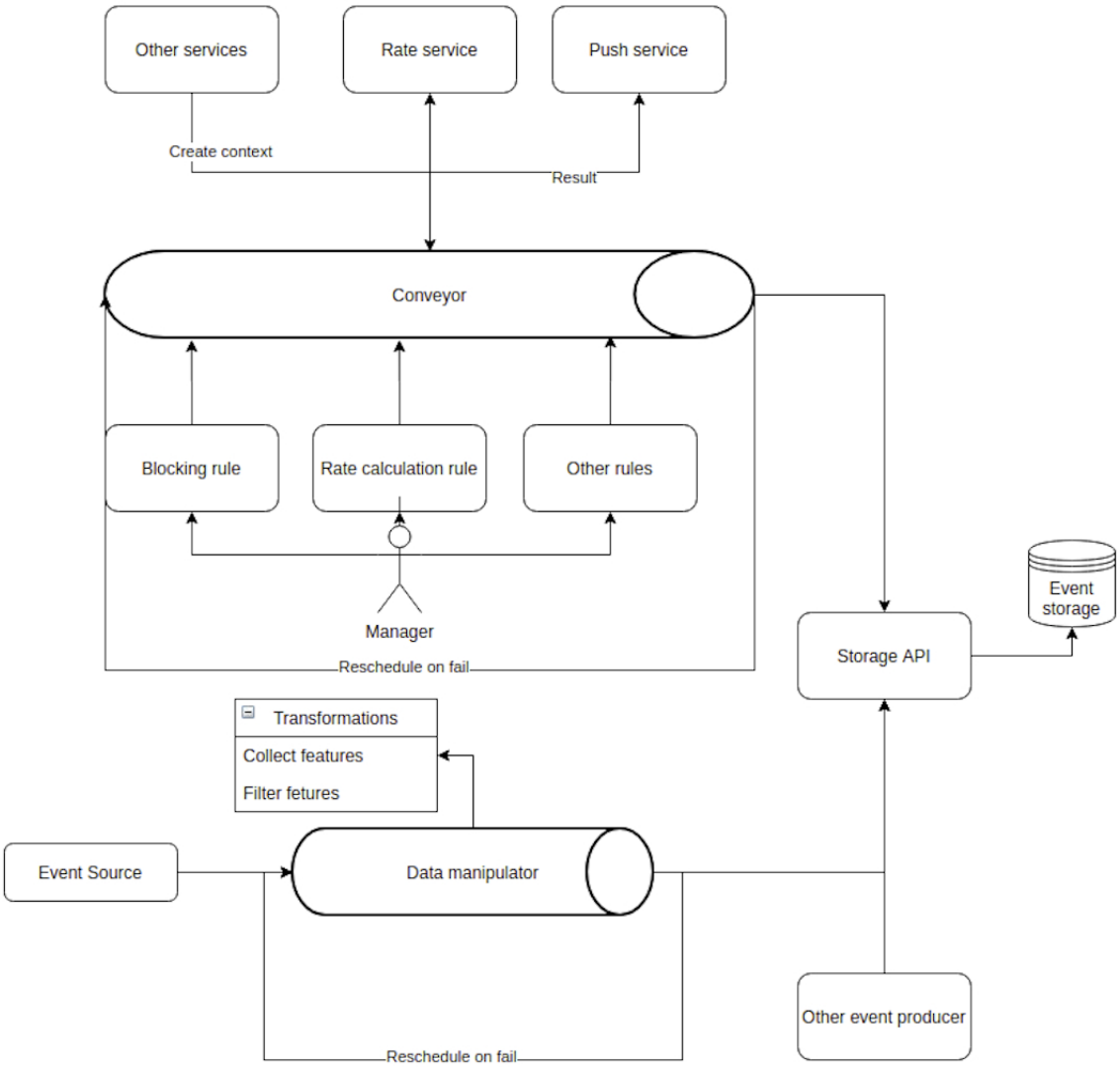

Рис. 6: Цикл обработки сообщений.

Глава 2. Прогноз рейтинга

2.1 Мотивация

Как уже упоминалось ранее, правила на json схемах не решают проблему масштабируемости, перечислим недостатки такого подхода:

- Гибко, но недостаточно
- Тяжело масштабируется

- Человеческий фактор
- Сложно обобщить правила на все условия

Решением в данной ситуации является математическая модель для прогноза начисления рейтинга, обученная с помощью алгоритмов машинного обучения. Так как рейтинг принадлежит промежутку от 0 до 100 и может быть только целочисленным, задачу можно сформулировать как задачу регрессии с округлением до целочисленного значения. Используемые обозначения:

X - множество объектов для обучения $x_1, x_2, \ldots, x_n$,

Y - множество возможных ответов алгоритма: $i \in [-10, 10]$,

y - вектор ответов для объектов из X,

Необходимо получить такой алгоритм из семейства алгоритмов a:

$$a(X, \theta) -> Y,$$

который позволяет достигнуть минимума на функции качества:

$$L(a(x_i, \theta) - y_i) \rightarrow_\theta min.$$

2.2 Постановка задачи

Исходные данные - 9 млн объектов. Выборка происходила с удалением дупликатов и обогащением случаями, по которым происходили обращения в службу поддержки из-за некорректных списаний рейтинга. Также поддержан баланс результирующего списания рейтинга - равномерное распределение объектов, результат регрессии которых от -10 до 10.

Определим признаковое пространство для множества объектов. Критерии выбора признака:

- Информативность

- Линейная независимость от других признаков

На основе этих критериев определено следующее признаковое пространство:

- тариф (эконом, бизнес и т.д.)
- прибыль водителя от заказа (RUB)
- отношение индекса кандидата к количеству кандидатов на заказ (всего 5 кандидатов, водитель третий в очереди, тогда значение 3/5)
- предполагаемая длительность поездки
- наименьшее расстояние между точкой А и точкой Б
- предполагаемое время пути от водителя до пассажира
- количество отказов водителя от предыдущего успешного заказа до текущего
- количество выполненных заказов в течение суток
- время предложения заказа внутри одних суток

В качестве результата регрессии на обучающей выборке будем использовать:

- Для принятых заказов - начисление рейтинга по предварительно настроенным менеджерами правилам
- Для неудачных заказов (все водители отказались) предполагаем, что нужно было предлагать 10 (максимизировать вероятность принятия заказа)

В качестве метрики используем MSE [17]:

$$Error = (y_i - y_{pi})^2, i \in [1, N]$$

2.3 Используемые алгоритмы

Рассмотрим наиболее популярные алгоритмы решения задачи регрессии, а именно:

1. Линейная регрессия
2. Градиентный бустинг

2.3.1 Линейная регрессия

Модель линейной зависимости представлена как:

$$\overline{y} = X * \overline{w} + \overline{e},$$

где $\overline{w}$ - вектор весов модели,
$\overline{e}$ - случайная ошибка, непрогнозируемая моделью.

Также для случая линейной модели должны выполняться следующие условия:

1. Матожидание случайных ошибок равно нулю
2. Дисперсия случайных ошибок конечная и имеет одинаковую величину
3. Случайные ошибки некоррелированы

Решение поставленной задачи сформулируем как минимизацию среднеквадратической ошибки методом наименьших квадратов:

$$L(X, y, w) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - w^T * x_i)^2 = \frac{1}{2n}||\overline{y} - X * \overline{w}||_2^2$$

Найдем решение, приравнивая производную к нулю:

$$\frac{\partial L}{\partial \overline{w}} = \frac{\partial}{\partial \overline{w}}\frac{1}{2n}(\overline{y}^T\overline{y} - 2\overline{y}^T X\overline{w} + \overline{w}^T X^T X\overline{w}) = \frac{1}{2n}(-2X^T\overline{y} + 2X^T X\overline{w}) = 0$$

$$-X^T\overline{y} + X^T X\overline{w} = 0 \Leftrightarrow \overline{w} = (X^T X)^{-1}X^T\overline{y}$$

Стоит отметить, что полученное аналитическое решение не существует в случае вырожденной матрицы X (определитель матрицы равен нулю) или имеет большой разброс значений, когда значение определителя приближается к нулю. Чтобы предотвратить возникновение подобных ситуация воспользуемся регуляризацией [11]. В общем виде она представлена так:

$$L_{reg}(X, y, w) = L(X, y, w) + l_R(w)$$

Рассмотрим два варианта реализации R(w):

$$R(w) = 1/2||w||_2{}^2 - L2 \text{ регуляризация}$$

$$R(w) = ||w||_1 - L1 \text{ регуляризация}$$

Решение для задачи с L2 регуляризацией находится аналитически в силу дифференцируемости заданной функции. Для решения задачи линейной регрессии с L1 регуляризацией следует использовать метод стохастического градиентного спуска, описанном в [15]. Он основан на идее градиентного спуска с использование ограниченного подмножества данных для вычисления градиента целевой функции.

2.3.2 Gradient boosting (catboost)

Метод градиентного бустинга уже давно зарекомендовал себя как метод, способный решить большое множество задач. Идея заключается в том, чтобы объединить некоторое число простых алгоритмов, которые демонстрируют результаты чуть лучше, чем случайный выбор, так, чтобы в ансамбле они давали гораздо меньшую ошибку. Семейство функций, которые будут использоваться для бустинга обозначим как:

$$f(x, \theta), x \in X,$$

θ - параметр функции

Тогда зависимость, которая должна быть найдена может быть записана

так:

$$f'(x) = f(x, \theta')$$

$$\theta' = argmin_\theta(E_{xy}[L(y, f(x, \theta))])$$

Аналитическое решение для получения оптимального значения θ существует редко, поэтому следует использовать итеративные алгоритмы нахождения точки минимума. Итоговая функция регрессии выглядит как линейная комбинация функций из заданного семейства функций. Находить оптимальное решение данной задачи затруднительно, поэтому в градиентном бустинге происходит жадное наращивание, каждый раз добавляя в сумму такое слагаемое, которое дает наиболее оптимальное значение функции потерь:

$$F_m(x) = F_m(x) + b_m * f(x, \theta')$$

$$Q = \sum(L(y_i, F_m(x_i))) \to min$$

Далее из использования метода градиентного спуска следует, что наибольшая выгода будет получена при добавлении нового слагаемого следующим образом:

$$F_m = F_{m-1} - b_m * grad(Q)$$

$$b_m = argmin_b(\sum(L(F_{m-1}(x_i) - b * grad(Q))))$$

Следует заметить, что градиент функционала ошибки представляет собой только вещественное число, тогда как ожидается найти функцию f(x, θ'). Решить эту проблему можно минимизируя функционал ошибки такого вида:

$$\theta' = argmin_\theta \sum(L(grad(Q), f(x, \theta)))$$

Подробнее про градиентный бустинг и его разновидности можно найти

в [13].

2.4 Результаты

Обучение модели линейной регрессии происходило с помощью пакета sklearn [16] на языке python 3.7 [4]. Для обучения градиентного бустинга использовалась библиотека catboost [6]. Анализ результата работы происходил с помощью кросс валидации с разделением исходной выборки на обучающую и тестовую в отношении 4 к 1. Результат представлен в таблице 3.

Таблица 2: Метрики обученных моделей

	Линейная регрессия без регуляризации	Линейная регрессия с L1 регуляризацией	Линейная регрессия с L2 регуляризацией	Градиентный бустинг (catboost)
MSE без округление результата	83736.81	70327.12	73117.78	13537.99
MSE с округлением результата	13742	9513	10064	2312

Стоит отметить, что распределение значений ошибок у всех обученных моделей примерно одинаковое и показано на рисунке 5.

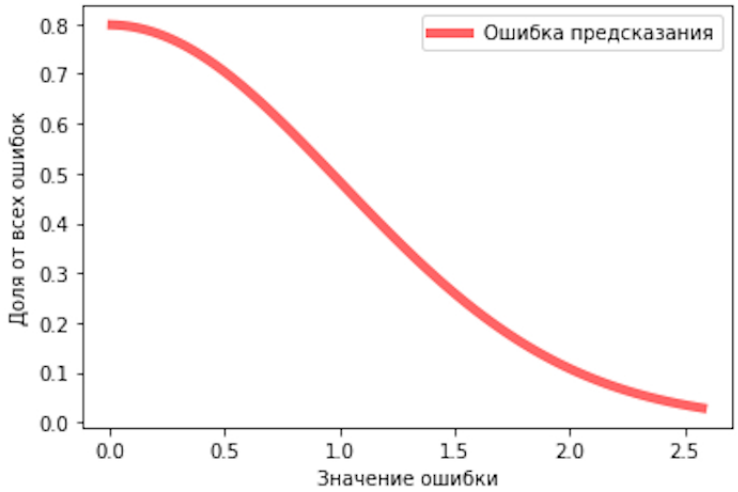

Рис. 7: Распределение ошибок.

Выводы

В рамках работы был разработан сервис обработки потока сообщений, способный выдержать предполагаемую нагрузку, для этого выбран наиболее подходящий способ коммуникации сервиса с базой данных. Был придуман формат для хранения правил, применяемых к сообщениям в обработке, который позволяет без изменений в коде создавать реакции на разные события при разных условиях. В результате этого скорость выполнения продуктовых требований сократилось с двух суток до 2 минут. Также были рассмотрены методы решения задачи регрессии. Лучший результат продемонстрировал метод градиентного бустинга, при этом допущенные ошибки незначительны с точки зрения бизнеса и результат можно интерпретировать близкий к идеальному.

Заключение

Сервис обработки потока сообщений, разработанный в рамках первой главы, успешно выполняет поставленную перед ним задачу. Выбранный формат генерации реакций на сообщения покрывает большое множество возможных использований. Возможностей языка python и хорошо продуманной архитектуры оказалось достаточно для выдерживания нагрузки кратной 10 от текущей, что позволяет за крайне небольшой промежуток времени обработать очередь в сотни тысяч событий. Лучший результат из рассматриваемых моделей для предсказания рейтинга за заказ показал метод градиентного бустинга, что вполне объяснимо. Задача интуитивно может быть решена путем построения дерева решений. К примеру, если рассматривать последнего кандидата на заказ, которых было 10, то необходимо предложить максимальный рейтинг за заказ, чтобы клиент не ушел к конкуренту.

В дальнешем возможны следующие улучшения работы сервиса:

1. Возможность задания реакции при помощи удобного UI с валидацией на исторических правилах
2. Динамическое обучение модели прогноза рейтинга
3. Возможность задавать различные условия для исторических событий и их комбинации (2 отмены заказа и 2 принятия заказа, где была хотя бы одна оплата банковской картой)

Список литературы

[1] Дональд Кнут Искусство программирования, том3. Сортировка и поиск. 2-е изд. М.: Вильямс, 2007.

[2] Основы операционных систем // intuit.ru URL: https://www.intuit.ru/studies/courses/2192/31/info (дата обращения: 23.01.2020).

[3] Приемы объектно-ориентированного проектирования. Паттерны проектирования / Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж., Спб.: Питер, 2001.

[4] About Python // python.org URL: https://www.python.org/ (дата обращения: 10.12.2019).

[5] A B M Moniruzzaman, Syed Akhter Hossain NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison // arXiv. 2013.

[6] CatBoost is a high-performance open source library for gradient boosting on decision trees // catboost.ai URL: https://catboost.ai/ (дата обращения: 10.05.2020).

[7] ECMA-404 The JSON Data Interchange Standard. // json.org URL: https://www.json.org/json-ru.html (дата обращения: 21.02.2020).

[8] Hong Zhu, Patrick A. . Hall, John H.R. May Software unit test coverage and adequacy // ACM Computing Surveys. 1997.

[9] Junaid Magdum, Ritesh Ghorse, Chetan Chaku, Rahul Barhate, Shyam Deshmukh A Computational Evaluation of Distributed Machine Learning Algorithms // IEEE.

[10] Kleppmann M. Designing Data-Intensive Applications. 1-е изд. Sebastopol: O'Reilly Media, 2017.

[11] Monique Borg Inguanez Regularization in Regression: Partial Least Squares and Related Models. Leeds: University of Leeds, 2015.

[12] PL/Python — процедурный язык Python // postgrespro.ru URL: https://postgrespro.ru/docs/postgresql/12/plpython (дата обращения: 12.02.2020).

[13] Open Machine Learning Course. Topic 10. Gradient Boosting // medium.com URL: https://medium.com/open-machine-learning-course/open-machine-learning-course-topic-10-gradient-boosting-c751538131ac (дата обращения: 05.05.2020).

[14] Paul C. Jorgensen, Carl Erickson Object-Oriented Integration Testing // ACM Computing Surveys. 1994.

[15] S. BhatnagarH.L. PrasadL.A. Prashanth Stochastic Recursive Algorithms for Optimization. London: Springer, 2013.

[16] scikit-learn Machine Learning in Python // scikit-learn.org URL: https://scikit-learn.org/stable/index.html (дата обращения: 20.05.2020).

[17] T. Chai, R. R. Draxler Root mean square error (RMSE) or mean absolute error (MAE)? – Arguments against avoiding RMSE in the literature // Geosci. Model Dev. 2014. №7.

[18] The Importance of Idempotence // antoineleclair.ca URL: http://antoineleclair.ca/2014/08/15/the-importance-of-idempotence/ (дата обращения: 20.02.2020).