

Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА/GRADUATION THESIS

Разработка высоконагруженной древовидной системы хранения рекламной  
статистики ВКонтакте

**Автор/ Author**

Безбородов Павел Андреевич

**Направленность (профиль) образовательной программы/Major**

Информатика и программирование 2017

**Квалификация/ Degree level**

Бакалавр

**Руководитель ВКР/ Thesis supervisor**

Корнеев Георгий Александрович, кандидат технических наук, Университет ИТМО,  
факультет информационных технологий и программирования, доцент (квалификационная  
категория "ординарный доцент")

**Группа/Group**

М3436

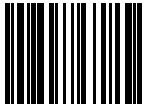
**Факультет/институт/кластер/ Faculty/Institute/Cluster**

факультет информационных технологий и программирования

**Направление подготовки/ Subject area**

01.03.02 Прикладная математика и информатика

Обучающийся/Student

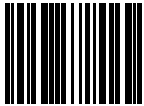
Документ подписан	
Безбородов Павел Андреевич	
05.05.2021	

(эл. подпись/ signature)

Безбородов  
Павел  
Андреевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/ Head  
of Graduate Project

Документ подписан	
Корнеев Георгий Александрович	
15.05.2021	

(эл. подпись/ signature)

Корнеев  
Георгий  
Александрович

(Фамилия И.О./ name  
and surname)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /  
OBJECTIVES FOR A GRADUATION THESIS**

**Обучающийся / Student** Безбородов Павел Андреевич

**Группа/Group** М3436

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет информационных технологий и программирования

**Квалификация/ Degree level** Бакалавр

**Направление подготовки/ Subject area** 01.03.02 Прикладная математика и информатика

**Направленность (профиль) образовательной программы/Major** Информатика и программирование 2017

**Специализация/ Specialization**

**Тема ВКР/ Thesis topic** Разработка высоконагруженной древовидной системы хранения рекламной статистики ВКонтакте

**Руководитель ВКР/ Thesis supervisor** Корнеев Георгий Александрович, кандидат технических наук, Университет ИТМО, факультет информационных технологий и программирования, доцент (квалификационная категория "ординарный доцент")

**Срок сдачи студентом законченной работы до / Deadline for submission of complete thesis** 31.05.2021

**Техническое задание и исходные данные к работе/ Requirements and premise for the thesis**

Требуется разработать распределенную систему для хранения рекламной статистики. Система должна удовлетворять требованиям по эффективности обработки и хранения данных, а также их целостности и консистентности с учетом предметной области.

**Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)/ Content of the thesis (list of key issues)**

1) Постановка задачи.

а) Структура рекламы ВКонтакте

б) Формулировка требований консистентности

2) Исследование возможных решений на основе существующих технологий.

а) Изучение иерархических баз данных

б) Изучение возможности реализации на реляционных базах данных (на примере PostgreSQL)

в) Выводы

3) Описание и реализация системы.

а) Описание схемы базы данных - структур возможных запросов и ответов

б) Объекты, их шардирование

в) Счетчики, их описание, настройка, способ хранения

- г) Агрегация значений и ее реализация
- д) Лимиты и дополнительные возможности системы
- е) Механизмы консистентности - по иерархии, по времени
- 4) Результаты
  - а) Общие результаты - о системе
  - б) Сравнительные результаты - сравнение со старым решением
- 5) Интеграция.

**Перечень графического материала (с указанием обязательного материала) / List of graphic materials (with a list of required material)**

**Исходные материалы и пособия / Source materials and publications**

- 1) Предыдущие системы хранения
- 2) Дейт К. Введение в системы баз данных
- 3) Joe Celko. Trees and hierarchies in sql for smarties.
- 4) UKEssays. (November 2018). The Hierarchical Model. Retrieved from <https://www.ukessays.com/essays/information-technology/hierarchical-data-model.php?vref=1>
- 5) Avi Silberschatz, Henry F. Korth, S. Sudarshan. Database System Models. <https://www.db-book.com/db6/index.html>

**Дата выдачи задания/ Objectives issued on 03.05.2021**

**СОГЛАСОВАНО / AGREED:**

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Корнеев Георгий Александрович	
03.05.2021	

(эл. подпись)

Корнеев  
Георгий  
Александрович

Задание принял к  
исполнению/ Objectives  
assumed by

Документ подписан	
Безбородов Павел Андреевич	
18.05.2021	

(эл. подпись)

Безбородов  
Павел  
Андреевич

Руководитель ОП/ Head  
of educational program

Документ подписан	
Станкевич Андрей Сергеевич	
18.05.2021	

(эл. подпись)

Станкевич  
Андрей  
Сергеевич

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**АННОТАЦИЯ  
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ /  
SUMMARY OF A GRADUATION THESIS**

**Обучающийся/ Student**

Безбородов Павел Андреевич

**Наименование темы ВКР / Title of the thesis**

Разработка высоконагруженной древовидной системы хранения рекламной статистики ВКонтакте

**Наименование организации, где выполнена ВКР/ Name of organization**

Университет ИТМО

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ/  
DESCRIPTION OF THE GRADUATION THESIS**

**1. Цель исследования / Research objective**

Разработка высоконагруженной древовидной системы хранения рекламной статистики

**2. Задачи, решаемые в ВКР / Research tasks**

а) Разработка системы хранения адаптированной под рекламную структуру соцсети ВКонтакте б) Достижение производительности достаточной для ускорения обработки рекламных взаимодействий в) Гарантирование консистентности важных данных

**3. Краткая характеристика полученных результатов / Short summary of results/conclusions**

Разработана новая система хранения рекламной статистики, успешно внедренная в эксплуатацию в социальной сети ВКонтакте. Применение системы упростило учет рекламных взаимодействий на сайте и уменьшило количество используемых ресурсов.

**4. Наличие публикаций по теме выпускной работы/ Have you produced any publications on the topic of the thesis**

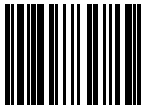
**5. Наличие выступлений на конференциях по теме выпускной работы/ Have you produced any conference reports on the topic of the thesis**

**6. Полученные гранты, при выполнении работы/ Grants received while working on the thesis**

**7. Дополнительные сведения/ Additional information**

Обучающийся/Student

Документ подписан	
----------------------	--

	
Безбородов Павел Андреевич	
05.05.2021	

(эл. подпись/ signature)

Безбородов  
Павел  
Андреевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/ Head  
of Graduate Project

Документ подписан	
Корнеев Георгий Александрович	
15.05.2021	

(эл. подпись/ signature)

Корнеев  
Георгий  
Александрович

(Фамилия И.О./ name  
and surname)

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	6
1. Обзор задачи, системы хранения и рекламной статистики «ВКонтакте» .....	7
1.1. Структура рекламы «ВКонтакте» .....	7
1.1.1. Термины .....	7
1.1.2. Взаимодействие с рекламой .....	8
1.2. Системы хранения ВКонтакте .....	8
1.2.1. Движки .....	8
1.2.2. TypeLanguage .....	9
1.2.3. Бинарный журнал событий .....	9
1.2.4. Снимок .....	9
1.2.5. Метафайлы .....	10
1.2.6. Шардирование .....	10
1.2.7. Запросы между движками .....	10
1.2.8. Кроны .....	11
1.2.9. Тестирование движков .....	11
1.3. Требования к движкам .....	11
1.3.1. Взаимодействие с объектами .....	11
1.3.2. Взаимодействие со счетчиками .....	12
1.3.3. Статистика .....	12
1.4. Требования к согласованности .....	13
1.5. Исследование возможных решений на основе существующих технологий .....	13
1.5.1. Предыдущее решение .....	13
1.5.2. Иерархические базы данных. IBM IMS .....	14
1.5.3. Реляционные базы данных. PostgreSQL .....	15
1.5.4. Базы данных временных рядов .....	16
1.6. Задачи, решаемые в работе .....	18
Выводы по главе 1 .....	18
2. Проектирование и реализация хранения и обработки рекламной статистики .....	19
2.1. Обзор архитектуры системы .....	19
2.2. Обзор архитектуры движка .....	20

2.3. Хранение данных .....	20
2.3.1. Настройки счетчиков .....	21
2.3.2. Объекты .....	22
2.3.3. Значения счетчиков .....	23
2.4. Реализация .....	24
2.4.1. Взаимодействие со счетчиками и объектами .....	24
2.4.2. Изменения счетчиков .....	25
2.4.3. Возвращение статистики .....	29
2.4.4. Активные объекты и периоды .....	30
2.5. Механизмы согласованности .....	30
2.5.1. Подтверждения запросов .....	31
2.5.2. Восстановление согласованности .....	33
2.5.3. Сверка исторических данных .....	35
2.5.4. Корректировка асимптотики запросов .....	35
Выводы по главе 2 .....	35
3. Тестирование, интеграция и сравнение полученной системы хранения с другими решениями .....	36
3.1. Тестирование .....	36
3.2. Интеграция .....	37
3.3. Сравнение .....	37
3.3.1. PostgreSQL .....	37
3.3.2. Предыдущее решение .....	38
Выводы по главе 3 .....	41
ЗАКЛЮЧЕНИЕ .....	43
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	44
ПРИЛОЖЕНИЕ А. Листинги .....	46
ПРИЛОЖЕНИЕ Б. Таблицы .....	49

## ВВЕДЕНИЕ

В современном мире существует множество Интернет-ресурсов, некоторые из них ежедневно посещают миллионы людей. Социальная сеть «ВКонтакте» имеет ежемесячный показатель активных пользователей равный 97 миллионам человек [3]. Вне сомнений, чтобы добиться работоспособности и надежности такой системы, необходимо качественное оптимизированное программное обеспечение для хранения и обработки данных. Исторически сложилось так, что большинство предоставляемых рынком решений показывали себя недостаточно эффективно в задачах «ВКонтакте». Для решения этой проблемы, социальная сеть разрабатывает узконаправленные системы хранения, которые могут быть использованы в условиях высокой нагрузки.

Реклама играет особую роль в развитии «ВКонтакте», является основным способом монетизации, а также позволяет платформе эффективно развиваться. Пользователям комфортно в среде, где реклама попадет в область их интересов, а предприниматели, которые готовы создавать контент, получают максимально релевантный отклик. В результате привлекаются новые пользователи, растет использование социальной сети. Именно поэтому для «ВКонтакте» важно создавать эффективную, удобную и функциональную рекламу.

Существует множество показателей, по которым рекламодатель может судить об эффективности своей кампании, например затраты и показы каждого рекламного объявления. Значения этих показателей необходимо эффективно хранить. Таким образом сформировалось требование к реализуемой мной системе хранения.

В первой главе формулируются требования к системе, производится обзор предметной области задания и возможных готовые решения, для некоторых из которых объясняется невозможность использования.

Во второй главе описывается реализация системы для хранения рекламной статистики.

В третьей главе производится сравнение результата работы с готовым и старым решениями.



## ГЛАВА 1. ОБЗОР ЗАДАЧИ, СИСТЕМЫ ХРАНЕНИЯ И РЕКЛАМНОЙ СТАТИСТИКИ «ВКОНТАКТЕ»

В разделе 1.1 производится обзор структуры рекламы «ВКонтакте» и взаимодействий с ней. В разделе 1.2 производится обзор систем хранения «ВКонтакте». В разделе 1.3 описываются требования к хранимым данным и запросам к ним. В разделе 1.4 описываются требования к согласованности данных. В разделе 1.5 производится обзор других решений. В разделе 1.6 перечисляются решаемые в работе задачи.

### 1.1. Структура рекламы «ВКонтакте»

#### 1.1.1. Термины

**Баннер** — элемент сайта с рекламой-ссылкой, отображается обычно под левым меню ВКонтакте.

**Лента новостей** — раздел ВКонтакте, где собраны все записи всех друзей и сообществ, на которых подписан пользователь. В них встраиваются рекламные записи.

**Рекламное объявление** — запись, которая будет появляться в баннере или ленте пользователя.

**Рекламная кампания** — набор рекламных объявлений, к которым могут применяться общие настройки.

**Рекламный кабинет** — соответствует одному пользователю и вмещает в себя все его кампании.

**Рекламный объект** — сущность, которой будет называться либо баннер, либо объявление, либо кампания, либо кабинет, либо же любой другой уровень этой иерархии, который рекламодатель пожелает сделать. Все они связаны по иерархии (отношение родитель — ребенок, пример: кампания это родитель нескольких объявлений).

**Счетчик** — число, которое отображает значение какого-либо показателя рекламного объекта. Например суммарное количество просмотров, количество уникальных просмотров или затраты. Счетчик разбит по времени и хранит значения на временных промежутках. Значения счетчиков в объекте, который является чьим-то родителем в основном является суммой значений этого же счетчика на детях.

**Лимит** — ограничение сверху на значение определенного счетчика определенного рекламного объекта на определенном отрезке времени.

**Квантование** — настройка определенного счетчика, округляющая его значение. Необходимо для пропуска маленьких значений, например копеек.

**Временной промежуток (период)** — это отрезок времени определенной длины, который обозначается его началом. Например, дневной промежуток 20 мая 2021 года.

**Тип временного промежутка (периода)** — это длина отрезка временного промежутка, например годой, суточный, пятичасовой и т.п.

**Таймфрейм** — это кортеж из объекта, счетчика, типа периода и самого периода. По сути это ключ значения счетчика, так как одно значение идентифицируется четырьмя этими идентификаторами.

### 1.1.2. Взаимодействие с рекламой

Основной точкой входа для рекламодателей является рекламный кабинет «ВКонтакте». В нем есть возможность создавать кампании и объявления, выставлять их настройки, включающие в себя аудиторию и ее фильтры и лимиты [4]. Примером лимитов может служить ограничение по затратам или количеству показов за промежуток времени: тратить на кампанию максимум 10000 рублей в день.

Для потребителей рекламы этой точкой являются различные элементы сайта, такие как лента новостей или баннеры. С этими элементами можно взаимодействовать, переходить по ссылкам, скрывать.

## 1.2. Системы хранения ВКонтакте

### 1.2.1. Движки

Для хранения данных в таких объемах, которые есть у «ВКонтакте», порой не подходят существующие решения. Например, СУБД MySQL [7] на начальных этапах развития социальной сети не выдерживала роста нагрузки [19]. Тогда было принято решение разрабатывать собственные системы хранения, которые были бы заточены под опеределенные виды данных, их назвали движками.

Движок — это бинарный исполняемый файл, который общается со внешним миром по протоколу Remote Procedure Call (RPC) [2]. Он хранит данные, учитывая их специфику и делая соответствующие оптимизации. А также исполняет только те запросы, которые необходимо для конкретных данных. «ВКонтакте» было разработано множество движков для хранения данных: сообщений, записей пользователей, таргетированной рекламы и многие другие.

Движки однопоточны, в основном из-за исторических причин и отсутствия поддержки многопоточности в общих технологиях и работе с сетью. Так как большинство запросов к движку легковесны, то затраты на синхронизацию будут соразмерны времени обработки самого запроса. Более того, большей параллельности можно добиться увеличивая число шардов в системе.

### **1.2.2. TypeLanguage**

TypeLanguage — это язык, на котором описывается интерфейс движка: процедуры, функции и структуры, которые могут являться их аргументами или же возвращаемыми значениями [16]. Эти описания используются для сериализации передаваемых по протоколу RPC сообщений. Каждому объекту, будь то функция или структура, присваивается уникальный идентификатор, именно они используются для десериализации входящих сообщений.

### **1.2.3. Бинарный журнал событий**

Для надежного сохранения важных данных необходимо использовать постоянное хранилище, исходя из которого можно будет восстановить последнее состояние системы после аварий, перезагрузок и обновлений. В качестве такого хранилища в движках используется бинарный журнал событий (бинлог). Удобно представлять бинлог в виде некоторого журнала, в котором в хронологическом порядке записаны все изменяющие состояние движка события. Бинлоги хранятся на дисках в бинарном формате в целях экономии памяти. При наивной реализации, для того чтобы восстановить состояние системы требуется прочитать и применить весь бинлог.

### **1.2.4. Снимок**

Подход с сохранением бинлога за всю историю работы движка, очевидно, имеет существенный недостаток. Такая система бесконечно накапливает данные, в результате чего они начинают использовать очень много ресурсов диска. Более того, при каждом перезапуске системы будет увеличиваться время старта, так как каждый раз при старте необходимо прочитать весь записанный бинлог, а он только растет.

В такой ситуации возможно использовать снимки. Снимок сохраняет состояние системы в момент работы, а не все события происходившие с ней. Снимки создаются по сигналу, система выделяет дочерний процесс который

записывает все необходимые данные в снимок, никак не задевая работу движка в основном процессе.

Снимки и бинлоги используют совместно, для восстановления состояния системы после перезапуска или сбоя используется последний снимок и та часть бинлога, которая была записана после его создания.

### **1.2.5. Метафайлы**

Невозможно хранить все данные, используемые во время работы движка, в оперативной памяти. Если накапливать исторические данные, то память будет исчерпана. Для того чтобы выгружать на диск те данные, запросы к которым происходят редко, используются метафайлы.

Метафайлы — это часть снимка, хранящиеся соответственно на диске, и которые могут по запросу быть подгружены в оперативную память. Подгрузка происходит в фоновом потоке и позволяет продолжать нормальную работу движка. Более того, обработка запросов, ожидающих подгрузки метафайлов, может быть отложена до их загрузки, в то время как движок продолжит обрабатывать следующие запросы.

### **1.2.6. Шардирование**

Для того чтобы хранить большое количество данных недостаточно одного экземпляра движка. В системе присутствует около 4,5 миллионов объектов и 200 миллионов установленных значений счетчиков, весь объем этих данных не уместится на одной физической машине. Поэтому все движки разделяются на кластера — набор серверов с несколькими запущенными на каждом шардами. Для того, что бы скрыть эту деталь реализации были созданы RPC-прокси.

Прокси обладают информацией о кластере: количестве серверов, запущенных шардов, их местоположении в сети. Эта информация описывается в конфигурационном файле кластера. Они также анализируют запрос и на основе полученной информации могут посылать запрос именно в необходимый шард, а в особых случаях даже в несколько шардов с последующей конкатенацией их ответов и выдачей наружу готового ответа.

### **1.2.7. Запросы между движками**

Иногда движкам необходимо общаться между собой, в частности, для того чтобы переслать запрос по иерархии объектов. Для этого реализован мо-

дуль с названием `grpc-queries`, который позволяет посылать такие запросы. Технология, как и прокси, основывается на конфигурационном файле кластера, в котором описаны данные о других движках.

Так как доступ к таким запросам необходимо ограничить, чтобы они не могли были быть посланы из бэкенда, существует отдельный `TypeLanguage` файл с описанием внутренних запросов, для которого не генерируются автоматические обертки на языке PHP [20]. А также в RPC-прокси отсутствует обработчик для такого рода событий.

### 1.2.8. Кроны

Крон — это периодически вызываемая функция, которая совершает фоновую работу. Если рассматривать детально, крон вызывается после каждого вызова функции `epoll_wait` (сотни раз в секунду).

Чаще всего кроны содержат какую-то информацию в очередях, которую нужно обработать не конкретно во время выполнения запроса, но по частям в течение некоторого времени после его обработки.

### 1.2.9. Тестирование движков

При создании движка есть возможность реализовать интеграционные и модульные тесты. Модульные тесты реализуются с помощью фреймворка `GoogleTest` [9] на языке C++ [15] и запускаются на каждую сборку. Интеграционные тесты проверяют сетевое взаимодействие с движком, а так как бэкенд сайта реализован на PHP 7 [20], на нем же реализованы и тесты.

В интеграционном тестировании локально поднимается несколько движков, которым посылают запросы и проверяют разного рода инварианты. При написании тестов также используется библиотека тестирования `PHPUnit` [1] разработанная ранее технология типизированной сериализации. `PHPUnit` позволяет контейнеризировать окружение и проверять инварианты движков.

## 1.3. Требования к движкам

Формальные требования к движку задавались командой рекламы «ВКонтакте». Они состояли из нескольких разделов.

### 1.3.1. Взаимодействие с объектами

- 1) Создание рекламного объекта с указанным идентификатором, родителем (если существует), настройками лимитов и показателем квантования. Идентификатор объекта имеет вид представленный в листинге 1.

Тип объекта обычно отображает его уровень в иерархии, а вектор используется для объектов с составным ключом.

- 2) Изменение свойств рекламного объекта по указанному идентификатору.
- 3) Получение дочерних объектов по идентификатору объекта.

Листинг 1 – Идентификатор объекта на языке C++

```
struct object_id {
    int object_type;
    std::vector<int> object_ids;
};
```

### 1.3.2. Взаимодействие со счетчиками

- 1) Создание счетчика с указанным идентификатором и перечнем настроек:
  - а) Агрегировать ли статистику по иерархии из детей.
  - б) Список сохраняемых промежутков времени, длительность их хранения и изменяется ли этот промежуток из запросов, или же может быть изменен только из ребенка.
  - в) Показатель округления значений счетчика (в целях отбрасывания маленьких значений).
- 2) Увеличение счетчика у указанного объекта по указанному промежутку времени.
- 3) Удаление счетчика, при условии отсутствия у него ненулевых значений во всех объектах.
- 4) Получение настроек счетчика.
- 5) Получение списка всех счетчиков.

### 1.3.3. Статистика

- 1) Получение значения по идентификатору счетчика, объекта и временному промежутку.
- 2) Получение всех значений счетчиков по идентификатору счетчика и объекта с возможностью фильтрации (по включению в промежуток или по типу промежутка).
- 3) Получение списка активных объектов - тех, значения счетчиков которых изменялись за последние сутки.
- 4) Получение списка активных временных промежутков, то есть тех у которых есть активные объекты.

### **1.4. Требования к согласованности**

Согласованность (целостность) данных — самая важная часть задачи, так как в числе счетчиков могут находиться денежные траты. Согласованность гарантируется относительно агрегации значений счетчиков в двух направлениях — по иерархии и по времени.

Агрегация значений происходит по правилам:

- 1) Для конкретного таймфрейма значение счетчика в родителе равно сумме значений этого же таймфрейма в детях.
- 2) Для конкретного объекта значение счетчика за период равно сумме значений этого же счетчика за периоды меньшей ширины, но входящих в него.

Для того чтобы гарантировать согласованность нужно предусмотреть механизмы, которые способны восстановить согласованность данных после падения как одной сущности системы, так и нескольких ее частей. А также уметь отменять запросы вдоль по иерархии, если на какой-то ее части запрос не смог примениться, как по причине аварии, так и по причине переполнения лимита.

Формально требования к согласованности были описаны следующим образом:

- 1) Запрос должен быть применен или отменен по всей иерархии в течение 20 секунд.
- 2) Данные после восстановления должны быть согласованы в течение 20 минут.
- 3) В случае утери бинлога за последние 20 минут, потерянные данные должны быть восстановлены.

### **1.5. Исследование возможных решений на основе существующих технологий**

В этой главе проводится анализ используемых на рынке решений и возможности их применения для удовлетворения требований.

#### **1.5.1. Предыдущее решение**

Ранее в качестве хранилищ для рекламной статистики использовался набор кластеров lists-engine [2]. Этот движок хранит сортированные списки

(многоуровнево) с полезной нагрузкой на каждый объект. Например, была возможность создать список объявлений, с подписком счетчиков, каждый из которых хранил бы список с соответствующими значениями по времени.

Основной проблемой такого решения было отсутствие согласованности между кластерами, отвечающими за разные части рекламной статистики. Просмотры хранились в одном кластере, денежные траты – в другом, а прочая статистика в третьем. Когда согласованность между ними нарушалась, необходимо было использовать специальный внешний механизм, способный ее восстановить. Этот механизм содержал сложную логику и из-за этого плохо поддерживался, что создавало еще больше проблем в случае его неисправностей. Он также запускался в течение суток, а так как некоторые показатели были рассинхронизированы в течение такого долго времени, это открывало некоторые уязвимости в платформе рекламы — возможно было показывать объявление практически бесплатно. Более того, почти на каждое взаимодействие с рекламой необходимо было выполнять запрос в несколько кластеров и согласовывать ответы из каждого из них.

Такое решение возникло спонтанно во время начального развития рекламной платформы «ВКонтакте», когда при проектировании системы не уделили достаточно внимания ее масштабируемости. Позднее эти недостатки стало очевидно, в результате чего и было сформировано задание по реализации специализированного движка для рекламной статистики, который бы сам гарантировал согласованность данных.

### **1.5.2. Иерархические базы данных. IBM IMS**

Так как множество объектов представляет из себя древовидную структуру, где существует связь ребенок-родитель, имеет смысл рассмотреть иерархическую модель данных в качестве возможной реализации системы, так как в ней данные представляются именно таким образом.

Наиболее известным представителем иерархических баз данных является IBM Information Management System [13], первая публикация которой была в 1968 году. На примере этой базы данных можно выделить преимущества и недостатки реализации системы на такой технологии.

Основными преимуществами являются:

- 1) Наиболее близкое к реальности представление иерархии объектов.
- 2) Транзакции, позволяющие применять запросы согласованно.



Недостатки, в свою очередь:

- 1) Возраст решений и слабая выразительность модели.
- 2) Сложность внедрения нового функционала.
- 3) Сложность реализации агрегации и проверки лимитов.
- 4) Проприетарная лицензия.
- 5) Сложность разработки на языке DL/I, разработанном IBM специально для IMS [13].
- 6) Сложность встраивания системы в существующую экосистему «ВКонтакте».

Резюмируя, реализация системы с использованием данного подхода является невозможной, в первую очередь из-за сложности реализации необходимых настроек на малоизвестных слабовыразительных API и практического отсутствия на рынке непроприетарных решений с продолжающейся поддержкой.

### **1.5.3. Реляционные базы данных. PostgreSQL**

Реляционные базы данных используют концепцию отношений, что позволяет им быть крайне гибкими и иметь возможность реализовать структуры соответствующие заданию. Более того, благодаря популярности реляционной модели, существует множество источников информации и ведется активная работа над расширениями, что делает эту модель еще более привлекательной.

Одним из самых популярных и мощных представителей реляционных баз данных является PostgreSQL [8], на его примере можно рассматривать возможную реализацию системы с помощью реляционной модели.

Для обоснования неэффективности реализации системы хранения на PostgreSQL, можно рассмотреть небольшую заготовку. Для реализации иерархии объектов в PostgreSQL достаточно добавить к каждому объекту внешний ключ, указывающий на его родителя [5]. Этого будет достаточно для нашего функционала, так как нам необходимо только повторить запрос в родителя. А также это будет эффективно по памяти, так как мы не будем хранить лишнюю информацию, например всю иерархию или множество ребер. На рисунке 1 приведена построенная физическая модель минимальной возможной реализации.

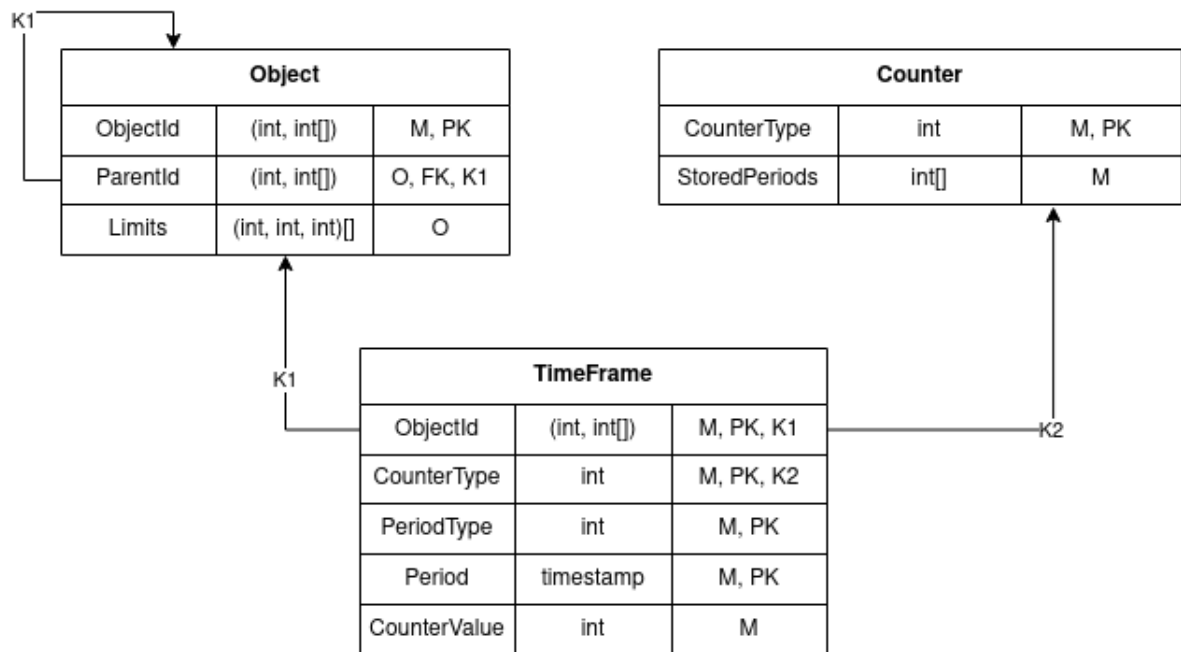


Рисунок 1 – Физическая модель системы хранения

В такой системе возможно хорошее шардирование данных силами PostgreSQL, когда все данные, касающиеся одного объекта, будут находиться в одном шарде.

Мною была реализована минимальная заготовка такой системы для сравнения эффективности с решением, реализованным собственными силами. В листинге A.1 в приложении приведен запрос, изменяющий счетчик объекта без учета лимитов, других настроек и без возвращения статистики из запроса, но с учетом агрегации по времени и иерархии.

Из недостатков такой системы сразу можно выделить сложность рекурсивной функции, которая несколько раз изменяет таблицу на каждое изменение счетчика. В дополнение к этому, выполнять запрос необходимо в repeatable read транзакции, так как неповторяющиеся чтения неприемлемы, из-за того что в запросах таблица частично изменяется и может произойти перезапись изменений.

Сравнение эффективностей данной минимальной реализации на PostgreSQL и итоговым решением проводится в главе 3.

#### 1.5.4. Базы данных временных рядов

Так как одной из частей задачи является хранение статистики с агрегацией по времени, то можно рассмотреть специализированные базы данных вре-

менных рядов. Они заточены под хранение метрик и могут эффективно возвращать значение какого-то счетчика за указанный период времени.

Примуществами таких баз данных считают:

- 1) Оптимизация для агрегации статистики по времени.
- 2) Как следствие, возможность быстро отдавать статистику по указанному периоду.
- 3) Упрощенная визуализация данных, так как в них в основном хранят метрики с датчиков, количество ошибок и тому подобное.

В качестве примеров таких СУБД можно рассмотреть одни из самых распространенных, например Prometheus [14] и InfluxDB [10].

Разработчики Prometheus в документации напрямую заявляют о том, что их решение не подходит для хранения данных, где нужна точность:

« If you need 100% accuracy, such as for per-request billing, Prometheus is not a good choice as the collected data will likely not be detailed and complete enough.» [14]

Так как итоговое решение хранит важную информацию о тратах денег, решение основанное на Prometheus не удовлетворяло бы требованием задания. Этот недостаток также касается и InfluxDB, она оптимизирована для того, чтобы выдерживать высокую нагрузку и стабильность, иногда жертвуя при этом полнотой данных [10].

Основным недостатком баз данных временных рядов является отсутствие операции `update`, вместо него используется `insert` с заменой по ключу. В большинстве случаев это обоснованно, так как такие базы данных хранят исторические данные, которые редко обновляются, так как имеют единственный источник (например, показания датчиков). Но в поставленной задаче это является минусом, так как в системе рекламной статистики именно обновление счетчика является самой частой операцией, которая происходит буквально на каждом взаимодействии пользователя с рекламным объявлением и вызывается множеством источников.

Другим недостатком является отсутствие возможности реализовывать функции и дополнительную логику на уровне системы. Для этого необходимо реализовывать абстракцию над системой хранения и частично данные хранить в других местах (например сведения о лимитах), что увеличивает сложность системы и, вероятнее всего, уменьшает ее эффективность.

Таким образом, базы данных временных рядов не удовлетворяют важным требованиям к системе, в основном согласованности, не оптимизированы под частые обновления счетчиков, а также слабо выразительны, что делает их неприменимыми к данной задаче.

### **1.6. Задачи, решаемые в работе**

В данной работе были поставлены и решены следующие задачи:

- 1) Провести анализ рекламы «ВКонтакте» и поставленной проблемы хранения рекламной статистики и возможных вариантов для решения проблемы.
- 2) Спроектировать хранение объектов, счетчиков и их настроек, а также интерфейс для работы с ними в рамках экосистемы «ВКонтакте».
- 3) Спроектировать и разработать механизмы для поддержания согласованности, удовлетворяющие требованиям, которые описаны в разделе 1.4.
- 4) Протестировать и интегрировать готовую систему хранения рекламной статистики.
- 5) Провести сравнение с другими возможными решениями.

### **Выводы по главе 1**

В этой главе был произведен обзор предметной области задачи, сформулированы требования для ее выполнения. Были проанализированы несколько возможных архитектур для реализации решения поставленной задачи с использованием готовых решений, предложенных на рынке, и обоснована неприменимость большинства из них. Также произведен обзор старого решения и необходимости его замены. И наконец, реализован минимальный функционал с помощью реляционной базы данных PostgreSQL для произведения сравнительного обзора с решением, выполненным в ходе работы над заданием.

## ГЛАВА 2. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ХРАНЕНИЯ И ОБРАБОТКИ РЕКЛАМНОЙ СТАТИСТИКИ

В разделе 2.1 производится обзор архитектуры системы в рамках экосистемы «ВКонтакте». В разделе 2.2 обзревается архитектура отдельно взятого движка и детали его реализации. В разделе 2.3 описываются способы хранения данных. В разделе 2.4 описываются запросы к движкам и как они обрабатываются. В разделе 2.5 рассказывается про механизмы согласованности и как они работают при сбоях.

### 2.1. Обзор архитектуры системы

Система должна поддерживать большое количество хранимых данных и запросов, обращающихся к ним. По этой причине кластер состоит из нескольких серверов (физических машин), на каждом из которых существует несколько запущенных шардов движка. Обращения к ним проходят через RPC-прокси, которая владеет информацией о распределении объектов по шардам, а также реализует дополнительную часть логики подтверждения запроса. Схематическое представление системы приведено на рисунке 2.

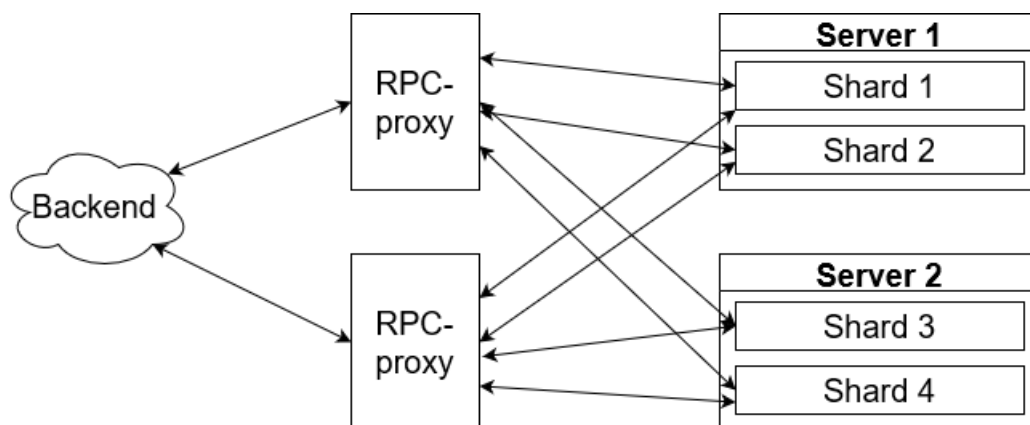


Рисунок 2 – Схематическое представление архитектуры системы

В системе всего две хранимых структуры — объекты и счетчики. Сами по себе счетчики не хранят какую-то информацию, кроме своего наличия и настроек, поэтому представлены в каждом шарде системы. Объекты, в свою очередь, случайно и равномерно распределены по шардам с помощью хеш-функции от идентификатора, адресат большинства запросов в движок определяется именно по этому идентификатору при помощи той же самой хеш-функции. Равномерное случайное распределение необходимо для балансиро-

вания шардов, так как чем выше объект находится по иерархии, тем чаще приходят его изменения, тем более нагружен шард, который его содержит.

## 2.2. Обзор архитектуры движка

Движку было дано название tree-stats, так как он хранит древовидную статистику. Он реализован на языке C++ [15], так как на нем реализовано большая часть общего кода, отвечающего за работу с сетью, бинлогом, снимками и прочими элементами движка. Бэкенд «ВКонтакте» работает на серверах с операционной системой Debian Linux [6], поэтому компиляция движков происходила с использованием стандартного для UNIX систем компилятора G++ из коллекции компиляторов GCC [17].

Исходя из требований, описанных в главе 1, а также системы шардирования описанной в предыдущей разделе, основными частями движка можно выделить: хранение объектов, реализацию запросов и механизмы согласованности.

Основными хранимыми данными являются настройки счетчиков, иерархия объектов и значения конкретных счетчиков. Большинство из них хранятся в оперативной памяти в хеш-таблицах, структурах удобных для поиска. Частично данные хранятся на диске при помощи метафайлов и подгружаются по запросу.

Сериализация запросов к движку реализована с помощью TypeLanguage [16] и автогенерируемого из него кода. На каждый запрос регистрируется функция-обработчик, которая вызывается при его получении. Благодаря продуманному способу хранения значений счетчиков, возможен быстрый поиск по значениям, их изменения и отдача сведений наружу.

Механизмы согласованности в основном представляют из себя кроны, которые производят сверку данных. Например собирают сумму значений на детях и проверяют, что собственное значение ему соответствует.

Подробнее три эти части описаны в следующих разделах.

## 2.3. Хранение данных

Для хранения данных используются две глобальные хеш-таблицы, одна хранит объекты, другая — настройки счетчиков. Хеш-таблицы позволяют эффективно по скорости осуществлять поиск объектов и счетчиков, с которыми необходимо производить операции [11]. В качестве реализации хеш-таблицы

был выбран стандартный для C++ `std::unordered_map`, функциональности которого достаточно для решения задачи. Сами значения счетчиков хранятся внутри объекта.

### 2.3.1. Настройки счетчиков

Так как исходя из требований задачи, множество счетчиков глобально, и каждый объект может иметь установленное значение любого счетчика, то все их настройки хранятся на каждом шарде.

В листинге 2 приведен код объявляющий структуру счетчика. В качестве полей объекта хранится показатель квантования, флаг необходимости агрегации по иерархии, количество созданных значений и настройки, описанные в разделе 1.3.

Листинг 2 – Объявление структуры счетчика

```
struct period_settings_t {
    int period_type;
    int periods_store_count;
    bool period_manual_change;
};

struct counter_settings_t {
    int64_t quantize_by;
    bool tree_manual_change;
    int number_of_objects;
    std::vector<period_settings_t> periods_types_settings;
};
```

Настройки представляют из себя перечисление хранимых типов периодов, для каждого из которых установлены:

- Количество хранимых периодов, периоды старше удаляются из системы. Это сделано в целях экономии памяти.
- Флаг, свидетельствующий о том, можно ли изменять этот тип период для этого счетчика вручную. Автоматически этот флаг установлен для наименьшего периода. Если флаг не установлен, то значения собираются со периодов меньшего размера.

Периоды и типы периодов представляются в удобном для отдела рекламы «ВКонтакте» виде. Тип периода это число из трех или четырех знаков, первые один или два знака которого отвечают за длину периода, а вторые два —

за базовую единицу исчисления. Возможные варианты базовых единиц в удобном формате представлены в таблице 1, в которой также определены форматы соответствующих периодов. То есть, тип периода равный 104 будет означать один день, 1502 — 15 минут, а 305 — три месяца. Самым частым используемым типом периода является 502 (пять минут), он оказался наиболее удобным для отдачи в режиме онлайн статистики для автоматического расчета стоимости показа на рекламном аукционе [18].

Таблица 1 – Возможные варианты базовых типов периодов

Базовая единица	Значение	Формат периода
секунда	1	YYYYMMDDHHmmss
минута	2	YYYYMMDDHHmm
час	3	YYYYMMDDHH
день	4	YYYYMMDD
месяц	5	YYYYMM
год	6	YYYY
за все время	7	1 (константа)

### 2.3.2. Объекты

Листинг 3 – Объявление структуры объекта

```
struct stat_object_t {
    object_full_id_t parent_full_id;
    std::unordered_set<object_full_id_t> children_full_id;
    std::unordered_map<int, counter_t> counters;
    int first_not_metafiled_timestamp = 0;
    jobs_metafile_t metafile;
    int metafile_id;
};
```

Объекты хранятся в единственном экземпляре, равномерно распределенными по всем шардам, на рисунке 3 схематично изображено возможное шардирование иерархий. Стрелкой обозначено отношение ребенок-родитель, которая направлена в сторону родителя, движение запроса возможно только по стрелкам, но разные объекты одной иерархии могут находиться на одном шарде.



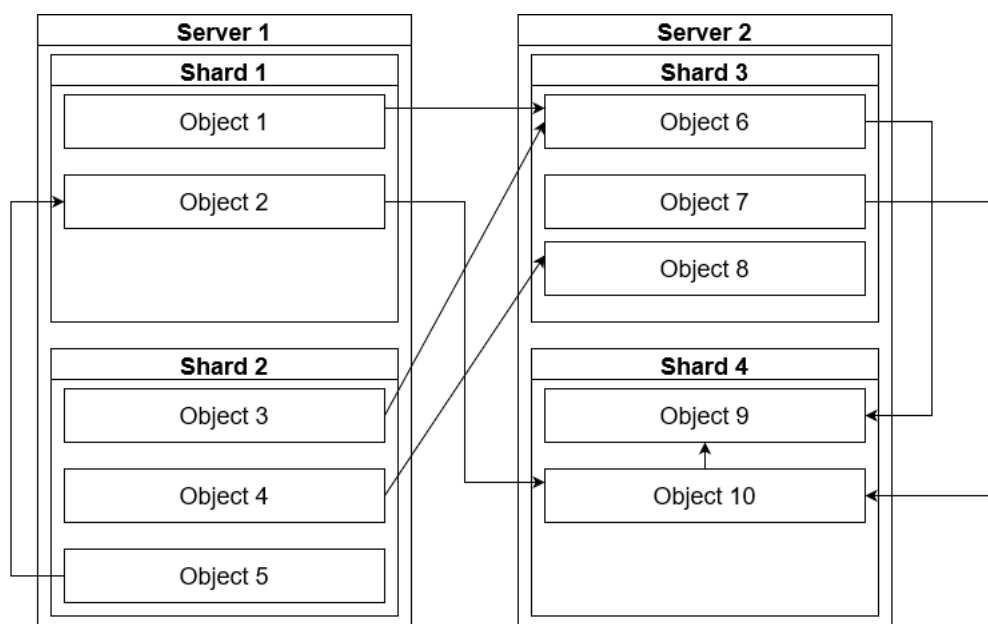


Рисунок 3 – Пример шардирования иерархии

Стоит отметить, что для каждого объекта определен свой метафайл. В нем хранятся значения счетчиков старше суток за типы периодов короче дней. То есть все периоды с типом 104 (дни) хранятся в памяти, а 103 (часы) — только за последние сутки. С целью правильно подгружать необходимые данные с диска также дополнительно хранится значение самой старой метки времени, хранящейся в памяти.

Объект также хранит не только идентификаторы своего родителя, но и о идентификаторы детей. Это необходимо для механизмов согласованности описанных в разделе 2.5, так как сверка происходит с помощью сбора значений счетчиков на детях.

### 2.3.3. Значения счетчиков

В каждом объекте реализована хеш-таблица, ключом которой является идентификатор счетчика, а значением — множество значений счетчика за разные периоды. Это множество значений представлено разряженным двумерным массивом, каждая строка в котором содержит значения определенного типа периода, а в конкретных ячейках записаны пары из самого периода и соответствующего ему значения. На рисунке 4 схематично изображено это представление.

Двумерный массив счетчиков отсортирован в обоих направлениях. Значения периодов при этом упрощаются до индекса, для того чтобы не хранить

неприятные рекламные представления. Индекс периода — это порядковый номер этого периода в UNIX-эпохе, например счетчик типа 106, соответствующий 2020 году, имеет индекс 50, так как UNIX-эпоха началась 1 января 1970 года.

107	(0, 10)					
106	(50, 3)		(51, 7)			
104	(18500, 1)	(18570, 2)	(18640, 1)	(18655, 2)	(18700, 1)	(18733, 3)

Рисунок 4 – Возможное представление значений счетчиков

В такой таблице удобно осуществляется поиск, разобранный в разделе 2.4.3 про запросы статистики. А данные хранятся эффективно по памяти, так как в таблице не хранятся пустые значения счетчиков и слишком старые счетчики выгружаются на диск.

Выгрузка на диск старых счетчиков необходима, так как в основном обновления присылают в относительно новые счетчики, хотя иногда бэкенд может подливать исторические данные.

Листинг 4 – Объявление структуры значений счетчика

```
struct counter_t {
    std::vector<std::vector<period_number_value_t>> value_vectors;
    std::vector<limit_t> limits;
    int64_t fraction = 0;
};
```

Как показано в листинге 4, рядом с таблицей значений также хранятся остаток от квантования и лимиты. Лимиты представляются как отсортированное множество пар из типа периода и его максимального значения, при каждом изменении счетчика происходит их проверка.

Несмотря на то, что лимиты создаются при создании объекта, они хранятся рядом с таблицей значений счетчиков, потому что лимит можно идентифицировать по объекту, счетчику и типу периода.

## 2.4. Реализация

### 2.4.1. Взаимодействие со счетчиками и объектами

Счетчик можно лишь создать или удалить.

Создать счетчик — значит задать его настройки и идентификатор. Идентификатор счетчика — это 32-битное целое число, а настройки описаны в разделе 1.3.1. При создании счетчика в движке создается новое вхождение в хеш-таблице счетчиков. Сам запрос посылается во все шарды движка силами RPC-прокси.

При этом удалить счетчик возможно, только если ни у одного объекта этот счетчик никогда не менялся. Такой подход обоснован тем, что, во-первых, иначе удаление было бы сложной операцией, требующей обхода всех объектов, а во-вторых, желанием сохранять все исторические данные.

Объект можно создать, изменить или перезадать. Удалить объект нельзя, это обосновано тем же требованием сохранять всю историю. При создании объекта, инициализируются внутренние структуры — таблица счетчиков и множество лимитов. А также новое вхождение в хеш-таблице объектов. При создании объекта важно сообщить его родителю о создании ребенка, чтобы он мог включить его в множество детей.

При этом настройки объекта — лимиты и квантования могут меняться, разумным может быть сценарий, когда пользователь увеличивает ограничение по своему объявлению после пополнения бюджета. Изменение объекта — это увеличение лимитов на указанную величину, а перезадание — это переопределение этих ограничений заново. Такое поведение реализовано исключительно по требованиям от команды рекламы.

При создании счетчиков и объектов, выполняется проверка конфликтов по идентификаторам. Если такой объект или счетчик уже существует — вернется ошибка.

#### **2.4.2. Изменения счетчиков**

Основным и самым часто посылаемым в движок запросом является изменение значения счетчика. Каждое изменение можно таймфреймом.

Алгоритм изменения выглядит следующим образом:

- 1) Локализовать нужный объект, проверить что он существует.
- 2) Локализовать нужные настройки счетчика, проверить, что он существует и в нем есть указанный тип периода.
- 3) При необходимости (изменение старого периода) подгрузить данные в память.

- 4) Найти и изменить все соответствующие периоду счетчики в таблице значений, если значение не присутствует в таблице значений, создать его.
- 5) При изменении/создании учитывать лимит, в случае если он переполнен досрочно завершить выполнение.
- 6) При наличии родителя отправить такой же запрос в него, предоставляя информацию о то, что этот запрос из ребенка.

В запросе изменения предусмотрена опция возврата новых значений, каждый запрос возвращает информацию о новых значениях своего счетчика у себя и всей иерархии над собой.

#### 2.4.2.1. Агрегация

Агрегация статистики должна происходить в двух направлениях — по иерархии и по времени. Само понятие агрегации формально можно описать следующим образом:

По иерархии:

$$V(O, C, T_p, P) = \sum_{O_C \in \text{child}(O)} V(O_C, C, T_p, P)$$

По времени:

$$V(O, C, T_p, P) = \sum_{P_i \in \text{periods}(P, T_{p-1})} V(O, C, T_{p-1}, P_i)$$

В приведенных формулах следующие обозначения:

- 1)  $V$  — функция, которая возвращает точное значение счетчика по его ключу – таймфрейму.
- 2)  $O$  — объект.
- 3)  $C$  — счетчик.
- 4)  $T_p$  — тип периода.
- 5)  $P$  — период.
- 6)  $\text{child}(O)$  — функция, возвращающая множество детей  $O$ .
- 7)  $\text{periods}(P, T)$  — функция, возвращающая множество периодов типа  $T$ , включенных в период  $P$ .

Агрегация по иерархии осуществляется следующим способом — запрос на изменение дублируется из родителя в ребенка, таким образом те же самые изменения применяются выше по иерархии.

Агрегация по времени происходит внутри одной таблицы значений счетчиков. Очевидно, при изменении определенного типа периода в таблице значений должны быть изменены все строки для типов больше или равных изменяемому, так как все столбцы отсортированы по типу периода, необходимо пройти по всем строкам от начала таблицы до строки изменяемого типа периода. Для изменения каждой строки необходимо значение периода обрезать до значения соответствующего строке типа. Обнаружение необходимого столбца выполняется бинарным поиском, так как все строки отсортированы по идентификатору периода.

Листинг 5 – Фрагмент исходного кода, отвечающий за агрегацию по времени

```
for (int i = period_order + 1; i < periods_types_settings.size();
    ++i) {
    if (!periods_types_settings[i].period_manual_change) {
        auto period_number_for_period_type = get_period_number(
            timestamp, periods_types_settings[i].period_type);
        auto value_for_period_type = get_value(
            period_number_for_period_type, counter->value_vectors[i])
        ;
        if (!check_limit(delta + value_for_period_type, fields_mask,
            periods_types_settings[i].period_type)) {
            applied = false;
        }
        increase_value(period_number_for_period_type, counter->
            value_vectors[i], delta);
    }
}
```

#### 2.4.2.2. Лимиты

Логика проверки лимитов специально реализована таким образом, что запрос применяется даже если итоговое значение превышает лимит, что видно в листинге 5. При этом общий ответ запроса вернет в RPC-прокси ответ о том, что запрос применился не полностью и по механизму согласованности без подтверждения запрос отменится в течение 20 секунд.

Так как лимиты заданы рядом с таблицей счетчиков, то их проверка не является сложной операцией. Лимиты хранятся отсортировано по типу периодов, поэтому производится бинарный поиск по типу периода в перечислении лимитов. И сверяется соответствующее значение.

### 2.4.2.3. Квантование

Квантование необходимо для отбрасывания незначущих единиц в счетчиках, проще говоря – это округление значений. В реализации отдельно от значения сохраняется остаток от деления на показатель квантования, остаток необходим для агрегации родителя. Для обоснования рассмотрим пример представленный на рисунке 5. В нем показатель квантования равен трем, при этом сумма остатков детей равна четырём, а значит значение счетчика в родителе должно быть больше на единицу.

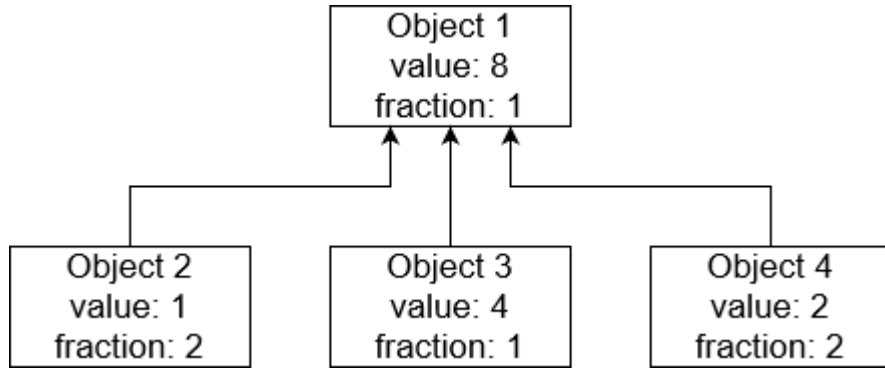


Рисунок 5 – Пример квантирования счетчиков. Схематично изображен только один таймфрейм в объекте

Рассмотрим временную асимптотику данной операции как самой частой, для поиска объекта необходимо найти его вхождение в хеш-таблице, по сложности это  $O(1)$  [11]. Далее ищется вхождение нужного счетчика в хеш-таблице счетчиков, что тоже имеет сложность  $O(1)$ . Для изменений всех периодов необходимо пройти по строчкам таблицы, количество которых константно для заданного счетчика, но обозначим его как  $c_{max}$ . В каждой строке выполняется бинарный поиск от общего числа объектов, который по сложности можно принять за  $O(\log_2 n)$  [12], где  $n$  — максимальное количество существующих счетчиков. Для проверки лимита на каждой операции осуществляется бинарный поиск по множеству лимитов, размер которого фиксирован для пары из объекта и счетчика и не может превышать  $c_{max}$ . Преобразование значения периода в другое по времени тоже ограничено константой. Итоговая асимптотика может быть вычислена следующим образом:

$$\begin{aligned}
 &O(1) + O(1) + O(c_{max}) \cdot O(\log_2 n) \cdot (O(\log_2 c_{max}) + O(1)) = \\
 &= O(c_{max}) \cdot O(\log_2 n) \cdot O(\log_2 c_{max}) = O(c_{max} \cdot \log_2 c_{max} \cdot \log_2 n)
 \end{aligned}$$

### 2.4.3. Возвращение статистики

Возвращение статистики существует двух видов. Первый вариант — возвращение конкретного значения по таймфрейму, выполняется поиском, сначала объекта, в нем таблицы счетчиков, далее в ней нужной ячейки. Он выполняется за  $(\log_2 c \cdot \log_2 n)$ , аналогично изменению, но не нужно проходиться по всем строчкам (только найти нужную бинарным поиском) и проверять лимиты. Второй – возвращение все счетчиков объекта, фильтруемых по счетчику и периоду, сначала выполняется по похожему сценарию, но возвращает всю таблицу счетчиков, которая еще сильнее разряжается с помощью фильтров.

Счетчики можно фильтровать двумя способами: задавать начало и конец отрезка идентификаторов, то есть например возвращать все счетчики с идентификатором от 5 до 55, или же указывать эти идентификаторы списком.

Периоды фильтруются сложнее, существуют некоторые показатели по которым происходит фильтрация:

- Тип периода
- Список или отрезок возвращаемых периодов (должны быть представлены в формате типа периода)
- Максимальное количество обрабатываемых счетчиков (чтобы ограничить время выполнения запроса)
- Максимальное количество возвращаемых счетчиков (чтобы ограничить размер ответа на запрос)
- Пагинация не является необходимой, но есть возможность указать последний выданный период, чтобы выполнить запрос снова и вернуть значения не уместившиеся в последние две настройки.

Рассмотрим ассимптотику запроса с фильтрацией по отрезкам. Локализация нужной таблицы выполняется за  $O(1)$ , так как это поиск нужного объекта в хеш-таблице [11]. Далее производится выделение нужных таблиц по списку счетчиков за  $O(c)$ , где  $c$  – количество необходимых счетчиков. Каждая таблица фильтруется поиском нужного отрезка строк за  $O(\log_2 c_{max})$  двумя бинарными поисками, так как таблица отсортирована по строкам. В каждой строке нужный отрезок локализуется за  $O(\log_2 n)$  такими же поисками внутри отсортированных строк. В итоге получаем:

$$O(c) \cdot O(\log_2 c_{max}) \cdot O(\log_2 n) =$$

$$= O(c \cdot \log_2 c_{max} \cdot \log_2 n)$$

#### 2.4.4. Активные объекты и периоды

Рекламе важно получать сведения об объектах и периодах с которыми происходили изменения, в основном потому что гарантии согласованности не мгновенны, а имеют некоторую задержку. Поэтому в системе они поддерживаются отдельно.

Активными периодами называются те, у которых есть хотя бы один ненулевой счетчик. Активными объектами называются те, у которых есть хотя бы один активный период.

Они хранятся отдельно в хеш-таблице активных периодов, где ключом выступает период, а значением — множество активных объектов. Это позволяет быстро получать и периоды, и объекты. Множества активных объектов пополняются во время получения запроса на изменение.

При этом для уменьшения размера ответа в запросах об активных элементах используются такие же фильтры, как и для получения статистики, описанные ранее.

#### 2.5. Механизмы согласованности

Основным требованием к системе является согласованность, в разделе 1.4 описаны общие требования к согласованности данных. Они реализуются несколькими механизмами с использованием кронов и имеют значительную общую часть.

Во всех кронах есть очередь элементов, которые необходимо обработать и какая-то необходимая для этого информация. У каждого элемента есть приоритет обработки — время после которого элемент должен быть обработан. Каждую итерацию крона из этой очереди достается несколько элементов (настройка при запуске движка), которые необходимо обработать, если их приоритет больше текущего времени, код итерации приведен на листинге 6. Элементы этой очереди добавляются или обновляются каждую операцию изменения.

Исходя из паттерна для реализации была выбрана интрузивная приоритетная очередь на минимуме, каждый элемент которой обратно связан с хеш-таблицей, ее объявление представлено в листинге A.2. Она позволяет:

- 1) Добавить элемент в очередь за  $O(\log_2 n)$  [11].



- 2) Изменить приоритет элемента за  $O(\log_2 n)$ , так как для поиска элемента в приоритетной очереди используется обратная хеш-таблица, из-за интрузивности это происходит за  $O(1)$  [11].
- 3) Получить элемент с наименьшим приоритетом за  $O(1)$  и удалить его за  $O(\log_2 n)$  [11].

Стоит отметить, что при большом размере очереди, в отличие от обычных деревьев поиска, сложность изменения приоритета будет меньше чем  $O(\log_2 n)$ , так как элемент вряд ли будет в верхних слоях приоритетной очереди, а его достаточно будет просто просеять вниз. В нашем случае это играет существенную роль, так как приоритеты часто обновляются.

Листинг 6 – Функция обработки элементов очереди

```
void check_next_batch() override {
    const int cur_timestamp = time(nullptr);
    for (int j = 0; j < top_check; j++) {
        if (check_queue.empty()) {
            return;
        }
        auto top_value = top();
        if (top_value.time_to_process > cur_timestamp) {
            return;
        }
        pop();
        process(top_value.object_to_process);
    }
}
```

### 2.5.1. Подтверждения запросов

Первым и самым важным является крон подтверждений. Он необходим для поддержания согласованности на уровне иерархии в рамках одного запроса и служит для восстановления целостности в случае сбоев сети, выпадением шарда и невозможности применить запрос, который переполняет лимит.

В качестве элементов очереди в этом кроне выступают запросы. В приоритетной очереди хранятся их уникальные идентификаторы, а в хеш-таблице сами запросы. После каждого изменения запрос записывается в эту очередь с приоритетом равным текущему времени + 20 секунд. Если запрос достается из очереди и обрабатывается, то он отменяется локальным выполнением обратного запроса. Если же в движок приходит подтверждение этого запроса, то он удаляется из очереди.

Рассмотрим подробнее выполнение запроса на изменение, которое схематично изображено на рисунке 6.

- 1) Из бэкенда в прокси приходит запрос на изменение объектов.
- 2) Прокси посылает каждый запрос в нужный шард.
- 3) Движок принимает запрос, выполняет изменение и записывает его в очередь.
- 4) Далее движок посылает запрос в шард с родителем.
- 5) Пункты 3 и 4 повторяются до последнего в иерархии объекта.
- 6) Из последнего шарда ответы собираются в один в обратном порядке и посылаются в прокси.
- 7) Прокси разбирает ответы, в случае успешности запросов — посылает подтверждение тем же шардам из пункта 2, а ответ отдает в бэкенд.
- 8) Каждый движок посылает подтверждение вверх по иерархии.

Таким образом, запрос отменится на всех участках иерархии, в случае если применение запроса было неудачным на любом участке и по любой причине: таймаут к следующему шарду, переполнение лимита или сбой в сети.

Важно отметить, что запросы на изменения можно отправлять пакетом в несколько штук. Так как ответ на пакет должен прийти один, то объединением ответов из разных иерархий также занимается прокси. В случае с подтверждениями, если хоть один запрос из пакета не применился, то отменить необходимо все запросы, именно поэтому подтверждение отсылает прокси, а не шард родителя.

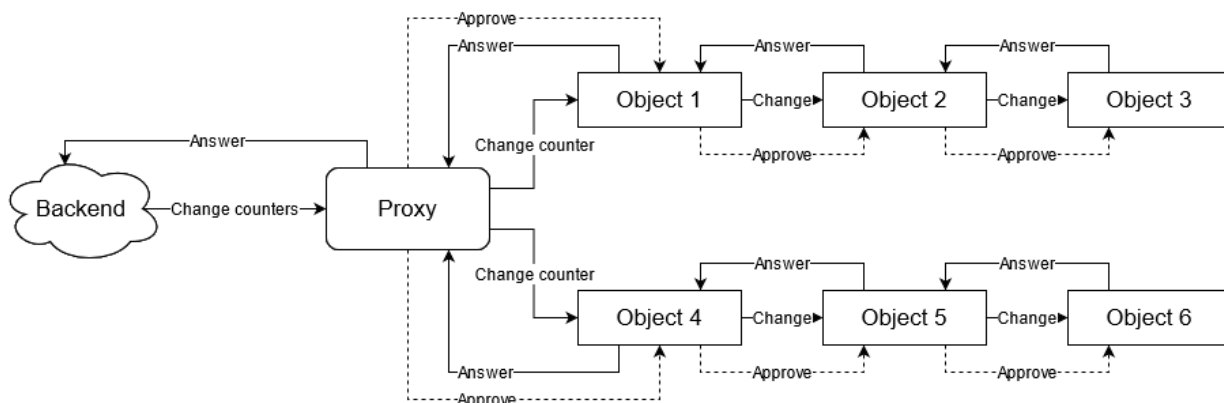


Рисунок 6 – Схема процесса работы крона подтверждений

Рассмотрим случаи сбоев, которые будут обработаны с помощью подтверждений:

- 1) Сбой шарда до запроса. Запрос не применится на этом шарде, предыдущий движок или прокси получит таймаут, а значит подтверждение не будет послано и запросы на предыдущих шардах отменятся.
- 2) Отказала сеть при отправке запроса в родителя. Можно считать шард родителя упавшим в рамках этого запроса, а значит восстановление произойдет таким же образом.
- 3) Сбой шарда после записи запроса в бинлог. Во время падения прокси получит плохой ответ от движков и не пошлет подтверждения. После восстановления движок применит запрос, но после таймаута отменит его.

### **2.5.2. Восстановление согласованности**

В случае когда цепочка подтверждений разрывается некоторые запросы могут быть случайно отменены. Иногда возможна ситуация, когда из-за физических неисправностей может быть потеряна большая часть бинлога одного шарда, в котором инициализировались какие-то счетчики. С целью исправления таких ошибок и восстановления пропавших счетчиков существует еще два механизма поддержания согласованности.

#### **2.5.2.1. Top-to-bottom крон**

В качестве элементов очереди в этом кроне выступают таймфреймы. В приоритетной очереди хранятся сами таймфреймы, а в хеш-таблице дополнительная информация: количество попыток финализироваться и множество детей. После каждого изменения счетчика его таймфрейм кладется в очередь (или обновляется приоритет этого таймфрейма) с приоритетом равным текущему времени + 20 минут. Если запрос достается из очереди, то начинается опрос детей — собирается сумма значений в детях по тому же таймфрейму, в случае ошибки значение исправляется. После проверки значение финализируется, если у объекта нет детей, то это происходит сразу. При опросе детей необходимо, чтобы их значения тоже были финализированы, если это не так — запрос откладывается, его приоритет в очереди увеличивается на 5 минут.

Таким образом top-to-bottom крон проверяет согласованность данных от нижних уровней иерархии к верхним (инициализируясь в обратном порядке). Выполняя проверки только после примерной уверенности финального значения счетчиков (таймаут в 20 минут).

Рассмотрим случаи восстановления после сбоев:

- 1) Не дошло подтверждение из ребенка. В таком случае при сверке данных объекта он проверит всех детей и соберет верную сумму, восстановив утерянный запрос.
- 2) Пропажа бинлога без потери счетчика, при этом пропадают изменения. Восстановление происходит таким же образом как в случае отсутствующего подтверждения.
- 3) Пропажа бинлога с потерей счетчика (потеря первого изменения). При опросе детей обнаружится, что счетчик не установлен, top-to-bottom его создаст и так же добавит в очередь крона.

Из последнего пункта возникает проблема, если счетчик пропал у объекта без родителя, то его никогда не создадут снова, так как родитель не опросит его. Для решения этой проблемы создан второй механизм.

#### **2.5.2.2. Bottom-to-top крон**

В очереди этого крона так же сохраняются таймфреймы. В качестве информации в хеш-таблице не хранится никакой полезной нагрузки, кроме индекса вхождения в приоритетную очередь (интрузивность). Элементы добавляются в очередь после их финализации в top-to-bottom, а когда обрабатываются – посылают специальный запрос инициализации top-to-bottom этого фрейма в родителя. Таким образом, если счетчик пропал, то он будет создан и через 20 минут восстановит свое значение.

#### **2.5.2.3. Выводы и дополнительные функции**

Эти механизмы могут гарантировать целостность данных согласно требованиям, описанным в разделе 1.4, но иногда являются тонким местом системы. В случае когда изменений накапливается слишком много, очереди кронов могут расти в размере слишком быстро и исчерпывать оперативную память. Для предотвращения ситуации когда память полностью исчерпана, существует ограничение на размер очередей, а также возможность аварийно сбросить очереди пожертвовав исправлением возможных ошибок. Стоит отметить, что в обычных условиях работы системы очереди не переполняются и последний механизм не используется, потенциально опасными были только массовые миграции на этапе интеграции движка.

### 2.5.3. Сверка исторических данных

В случае сбросов очередей кронов возможна утеря согласованности. В таких случаях отделом рекламы требуется восстановить ее хотя бы на исторических данных, которые хранятся в системе за все время, а конкретно это все значения с типом периода больше 104 (дни).

Для этого существует более периодичный `rescheck` крон, который запускается раз в неделю и добавляет в свою очередь все таймфреймы с периодами за последнюю неделю только с типами периодов больше или равными дням. Его логика обработки элементов абсолютно идентична `top-to-bottom` крону. Основное отличие от `top-to-bottom` заключается в том, что никакие счетчики не загружаются с диска, так как статистика по дням и более всегда хранится в памяти.

### 2.5.4. Корректировка асимптотики запросов

Так как во время обработки каждого запроса на изменение таймфрейм дополнительно добавляется или обновляется в трех очередях кронов, необходимо добавить к асимптотике  $O(\log_2 m)$ , где  $m$  — максимальный размер очереди, так как и вставка, и операция просеивания вниз имеют такую сложность [11]. Так как в очереди периодически уменьшается, то предположить ее реальный размер относительно общего количества счетчиков невозможно.

Итоговая асимптотика изменения:  $O(\log_2 m + c \cdot \log_2 c \cdot \log_2 n)$

## Выводы по главе 2

В этой главе была описана архитектура и реализация движка, как именно хранятся данные, как происходит обработка запросов и как была гарантирована хорошая согласованность.

### **ГЛАВА 3. ТЕСТИРОВАНИЕ, ИНТЕГРАЦИЯ И СРАВНЕНИЕ ПОЛУЧЕННОЙ СИСТЕМЫ ХРАНЕНИЯ С ДРУГИМИ РЕШЕНИЯМИ**

В разделе 3.1 описывается тестирование системы. В разделе 3.2 разбирается процесс интеграции новой системы. В разделе 3.3 производится сравнение с другими реализациями системы.

#### **3.1. Тестирование**

Большинство структур данных, которые использовались при реализации движка сложно отделимы от общей логики. Например, интрузивная очередь кронов требует от объектов возможности быть сериализованы силами TL. Таблица счетчиков и их изменение сильно завязаны на данных из запроса изменения. По этим причинам модульное тестирование не было реализовано.

Однако, было реализовано множество интеграционных тестов на языке PHP 7 [20] с использованием библиотеки PHPUnit [1]. В первую очередь, тесты проверяют инварианты системы: хранение объектов, счетчиков, выполнение базовых запросов, индексацию, подгрузку и выгрузку метафайлов для старых счетчиков. В дополнение к этому, проверяются некорректные запросы в движок.

Более сложная логика реализовывалась при тестировании механизмов согласованности: симулировалась утеря бинлога и выпадение движков. Для такой симуляции в тестах запускалось несколько движков и 1 прокси, создавалась иерархия, счетчики и посылались какие то запросы. Выпадение движка производилось простым отключением одного из шардов на время, после восстановления значения приводились в согласованное состояние. Для симуляции утери бинлога вместо выпавшего движка поднимался движок с пустым бинлогом и на нем создавались пропавшие объекты без счетчиков, такой сбой тоже обрабатывался корректно.

В дополнение, были реализованы стресс тесты для проверки стабильности системы при высокой нагрузке. В таких тестах поднимается до 20 шардов, выстраивается большая иерархия и в течение 5 минут она хаотично изменяется.

В целях упрощения тестирования и преждевременной локализации непредвиденных ошибок был создан тестовый кластер движка. В тестовый кластер реплицируется часть запросов из основного, процент отсылаемых запросов настраиваемый. Реализация тестового кластера возможна, благодаря

тому, что операции удаления невозможны значимых данных невозможны. Читающие запросы в тестовый кластер тоже отправляются, но на их результат никто не обрабатывает. При старте тестового кластера необходимо послать запросы на создание всех счетчиков, для этого существует отдельный скрипт.

Благодаря тому что тестовый кластер работает постоянно, он обеспечивает почти полное покрытие кода, более того, ошибки и сбои на нем не являются критическими, но их можно обнаруживать и исправлять.

## 3.2. Интеграция

Интеграция движка производилась в четыре этапа. Сначала происходила миграция старых данных из движков `lists` в `tree-stats`, для этого силами команды рекламы «ВКонтакте» был реализован специальный скрипт миграции. При его работе иногда наблюдались переполнения памяти движками, так как поток вливаемых данных был слишком большим и очереди заполнялись большим количеством элементов, ожидавших обработки. С целью предотвратить это была введено ограничение на размер очередей, скрипту же была добавлена задержка с целью замедлить поставку данных.

Вторым этапом была настройка репликации запросов в оба хранилища, бэкенд «ВКонтакте» посылал одинаковые по смысловой нагрузке запросы в `lists` и `tree-stats`. При этом происходила кросс-валидация, которая была затруднена наличием синхронизации `lists`. В старом движке данные почти не имели гарантий согласованности, она восстанавливалась в течение суток. Тем не менее, сверка исторических данных показывала совпадение результатов. На втором этапе при расчете данных и отдаче статистики наружу использовалось проверенное старое решение.

В третьем этапе отдача статистики переводилась на новое решение, но старое поддерживалось на случай непредвиденных ситуаций, чтобы в любой момент его можно было снова использовать.

После проверки работоспособности `tree-stats`, он стал основным решением. Запросы при этом перестали реплицироваться в `lists` и от его использования и поддержки, в рамках задачи хранения рекламной статистики, отказались.

## 3.3. Сравнение

### 3.3.1. PostgreSQL

В разделе 1.5 был произведен обзор возможных реализаций системы с использованием готовых систем. В частности, был разработан минималь-

ный работающий прототип с использованием СУБД PostgreSQL. Так как рассматриваемая реализация удовлетворяет минимальным требованиям функциональности, а потенциально может удовлетворить их все, в виду гибкости модели, то сравнение реализаций будет проводится по эффективности систем..

Для сравнения был написан скрипт на языке PHP 7, строящий одинаковую иерархию на обеих системах и выполняющий некоторое количество обновлений счетчиков, реплицировавшееся в обе системы и сравнивалось время выполнения запроса. Дополнительно, периодически выполнялся запрос получения статистики и сверка результатов, который не давал расхождений. Часть скрипта, генерирующая иерархию приведена в приложении в листинге А.3.

Входными данными для сравнительного скрипта были: количество слоев в иерархии ( $L_{count}$ ), объектов в слое ( $L_i$ ), изменений в одном запросе ( $R_{batch}$ ). Количество запросов было выбрано равным 10000, а количество счетчиков равным 10. Количество запросов не влияет на среднее время ответа, а счетчик в который посылается изменение выбирается случайным образом. В качестве типов хранимых периодов были выбраны час, день, месяц, год и «за все время».

Все результаты собраны в удобном виде в приложении в таблице Б.1.

Исходя из собранных результатов, реализованное решение, описанное в главе 2 оказалось как минимум в 4 раза быстрее решения на PostgreSQL. Именно время отклика является ключевым показателем для системы при сравнении, так как стабильность и согласованных данных являются абсолютными факторами, которые не могут быть сравнены, а только провалидированы.

### 3.3.2. Предыдущее решение

Несмотря на то, что основным обоснованием необходимости нового решения являлось упрощение инфраструктуры всего бэкенда с помощью встраивания механизмов синхронизации счетчиков в сам движок, повышение общей эффективности системы в некоторых аспектах также является важным показателем успешности.

Локальное сравнение предыдущего решения с новым является очень трудоемким процессом. Как уже было описано в разделе 1.5, старое решение представляет из себя набор кластеров, сложно согласованных друг с другом. За время работы старого решения, была собрана некоторая статистика, которая позволяет объективно провести сравнение с текущим решением.



Основным достоинством можно выделить общее количество серверов, необходимых для решения, а также их характеристики. Подробное сравнение физических показателей приведено в таблице 2

Таблица 2 – Сравнение реализаций

–	lists	tree-stats	Выигрыш
Количество серверов	54	5	$\sim 11$
RAM (GB)	3873	1290	$\sim 3$

Помимо общего количества необходимых ресурсов, снизилось также их потребление. На рисунке 7 представлены графики потребления оперативной памяти, которое снизилось в 1,4 раза. А на рисунке 8 график потребления дисковой памяти, снизившегося в три раза.

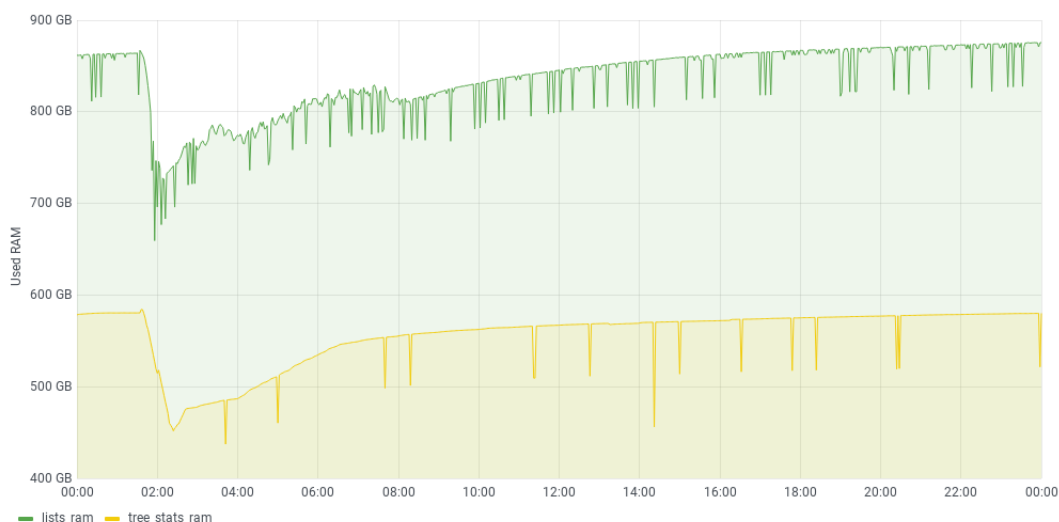


Рисунок 7 – Потребление оперативной памяти. Зеленый – lists, желтый – tree-stats

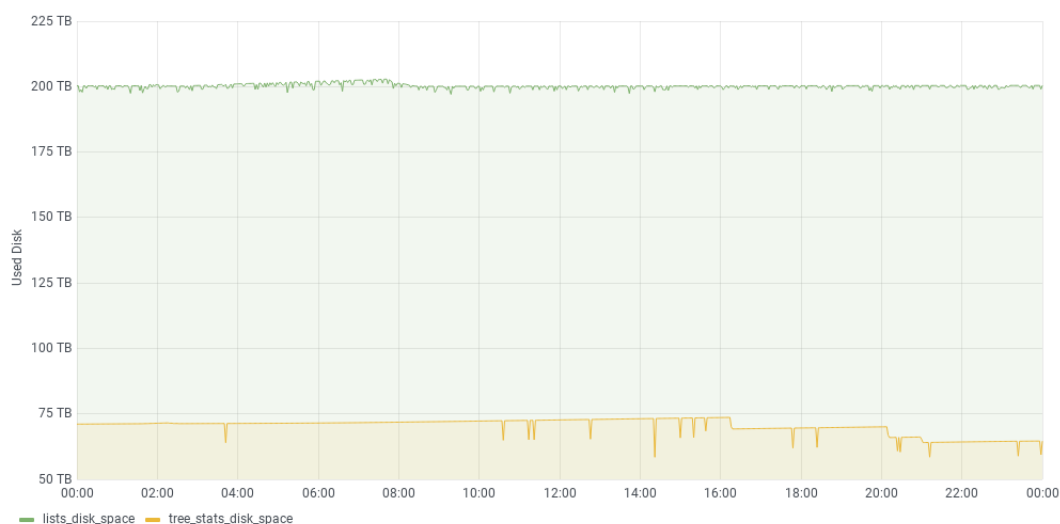


Рисунок 8 – Потребление дисковой памяти. Зеленый – lists, желтый – tree-stats

Существенным показателем улучшения производительности может служить среднее время ответа движков. В среднем видно уменьшение времени ответа на запрос в два раза, показанное на рисунке 9.

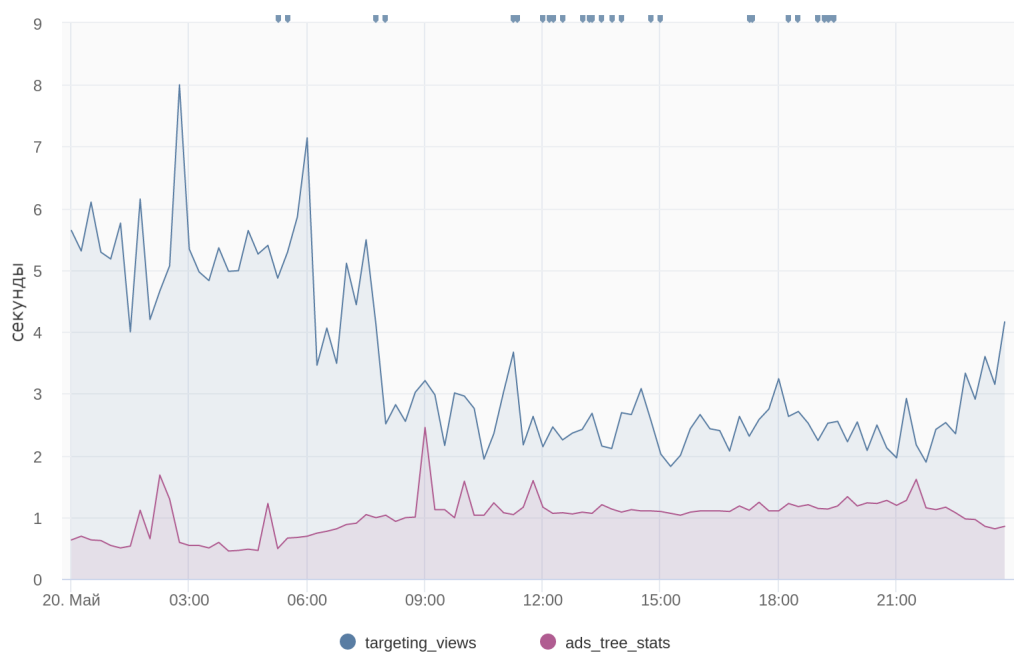


Рисунок 9 – Среднее время ответа от движка. Светлый – lists, темный – tree-stats.

Стоит отметить и улучшенную утилизацию диска, которая в среднем уменьшилась в 1,3 раза. Утилизация диска — это показатель нагруженности диска, который показывает относительное количество времени, когда диск

производил какие-либо операции, чем выше этот показатель тем скорее диск приходит в негодность. Сравнительный график приведен на рисунке 10.

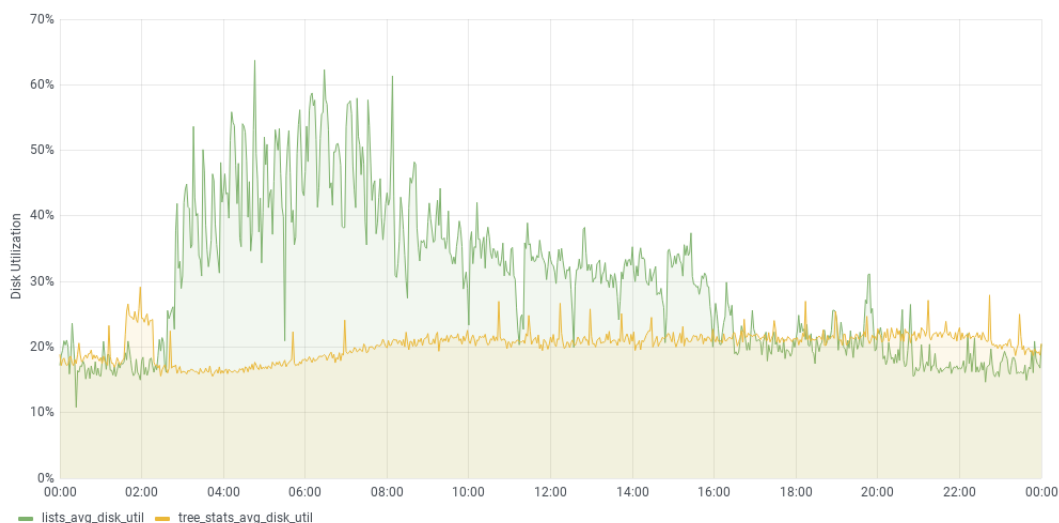


Рисунок 10 – Процент утилизации диска. Зеленый – lists, желтый – tree-stats

На графиках видны ночные аномалии, начинающиеся около 2 часов ночи. Эти аномалии объясняются процессом индексации (создания снимка). Во время создания снимка происходит много операций с диском, а потребление оперативной памяти спадает, так как большинство данных записывается на диск, например хранимые значения счетчиков, о которых рассказано в разделе 2.3.2.

Если рассуждать о теоретически возможных затратах ресурсов и времени, то сравнение становится проводить еще сложнее. По причине того, что в новом решении все механизмы поддержания согласованности встроены в сам движок, то сравнивать их с внешними старыми не представляется возможным. Вероятнее всего, из-за расположения этих методов ближе к данным и учета этих механизмов при изначальном проектировании системы, общее количество затрачиваемых на поддержание согласованности ресурсов было уменьшено.

### Выводы по главе 3

В данной главе были описаны процессы тестирования и интеграции системы, а также приведены результаты работы и сравнение решения с другими возможными. Заготовка на PostgreSQL показала себя хуже, даже с частично нереализованным функционалом. Старое решение, по проведенным исследо-

ваниям, оказалось менее эффективным по некоторым показателям, даже опуская необходимость дополнительных внешних ресурсов для синхронизации.

## ЗАКЛЮЧЕНИЕ

В рамках этой работы была спроектирована и реализована система хранения рекламной статистики социальной сети «ВКонтакте».

Внедрение нового решения позволило значительно упростить бэкенд «ВКонтакте», путем переноса поддержания согласованности внутрь движка. Более того, было значительно повышено качество кода учета рекламной статистики, потому что хранилище стало централизованным, отпала необходимость обращаться к множеству разрозненных кластеров.

Стоит отметить, что благодаря эффективной системе хранения, появилась возможность сохранять счетчики за малые промежутки времени, например по 5 минут, чего не было возможности осуществить ранее. Благодаря сохранению таких промежутков времени, новая система хранения позволяет получать в режиме онлайн траты и клики в течение дня, для использования в алгоритме просчета ставки объявления в рекламном аукционе, который также используется «ВКонтакте».

С точки зрения эффективности, новое решение позволило добиться увеличения скорости ответа на запросы и уменьшения утилизации диска, потребления оперативной и дисковой памяти. Более того, оно показало себя быстрее, чем возможная реализация на PostgreSQL.

В качестве дальнейших улучшений рассматриваются поддержка реплицирования в системе для улучшения стабильности и возможности быстро заменять отказавшие шарды на реплики (технология hot-swap), а также усложнение логики шардирования объекта с целью сбалансировать слишком большие объекты.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 *Bergman S.* Руководство по PHPUnit [Электронный ресурс]. — 2021. — URL: <https://phpunit.readthedocs.io/ru/latest/>.
- 2 *ВКонтакте.* Исходный код движков в открытом доступе [Электронный ресурс]. — 2021. — URL: <https://github.com/vk-com/kphp-kdb>.
- 3 О «ВКонтакте» [Электронный ресурс]. — 2021. — URL: <https://vk.com/about>.
- 4 О таргетированной рекламе «ВКонтакте» [Электронный ресурс]. — 2021. — URL: [https://vk.com/ads?act=office\\_help](https://vk.com/ads?act=office_help).
- 5 *Celko J.* Trees and Hierarchies in SQL for Smarties. — Morgan Kaufmann, 2012.
- 6 Debian Linux Official Site [Электронный ресурс]. — 2021. — URL: <https://www.debian.org/>.
- 7 *DuBois P.* MySQL. — Addison-Wesley Professional, 2013. — P. 1–12. 1153 p.
- 8 *Ferrari L., Pirozzi E.* Learn PostgreSQL. — Packt, 2020.
- 9 *Google.* GoogleTest open-source code [Электронный ресурс]. — 2021. — URL: <https://github.com/google/googletest>.
- 10 InfluxDB design insights and tradeoffs [Электронный ресурс]. — 2021. — URL: [https://docs.influxdata.com/influxdb/v1.8/concepts/insights\\_tradeoffs/](https://docs.influxdata.com/influxdb/v1.8/concepts/insights_tradeoffs/).
- 11 Introduction to Algorithms / T. H. Cormen [et al.]. — The MIT Press, 2011. — P. 151–159, 253–285, 348–355. 1292 p.
- 12 *Knuth D.* The Art of Programming. — Addison Wesley, 1998. — P. 410–415.
- 13 *Long R., Hain R., Harrington M.* IMS Primer. — IBM Redbooks, 2020.
- 14 Prometheus Overview [Электронный ресурс]. — 2021. — URL: <https://prometheus.io/docs/introduction/overview/>.
- 15 *Stroustrup B.* The C++ Programming Language. — Addison-Wesley, 2013.

- 16 *Telegram*. TL Language [Электронный ресурс]. — 2021. — URL: <https://core.telegram.org/mproto/TL>.
- 17 Using the GNU Compiler Collection / R. M. Stallman [et al.]. — GNU Press, Free Software Foundation, 2003. — Vol. 4, no 02.
- 18 *Yakovleva D., Popov A., Filchenkov A.* Real-Time Bidding with Soft Actor-Critic Reinforcement Learning in Display Advertising // 2019 25th Conference of Open Innovations Association (FRUCT). — 2019. — P. 373–382. — DOI: 10.23919/FRUCT48121.2019.8981496.
- 19 *Акулович А.* FAQ по архитектуре и работе ВКонтакте [Электронный ресурс]. — 2019. — URL: <https://habr.com/ru/company/oleg-bunin/blog/449254/>.
- 20 Руководство языка PHP [Электронный ресурс] / М. Achour [et al.]. — 2021. — URL: <https://www.php.net/manual/en/index.php>.

## ПРИЛОЖЕНИЕ А. ЛИСТИНГИ

Листинг А.1 – Простейший вариант функции, увеличивающей счетчики в PostgreSQL

```
create or replace function changePtoPT(PType int , P timestamp)
returns timestamp
as $$
begin
    return case
        when PType = 103 then date_trunc('hour', P)
        when PType = 104 then date_trunc('day', P)
        when PType = 105 then date_trunc('month', P)
        when PType = 106 then date_trunc('year', P)
        else '19700101'
    end;
end;
$$ language plpgsql;
```

```
create or replace procedure changeConcreteCounter(O object_id ,
    CType int , PType int , P timestamp , Val int)
as $$
begin
    insert into TimeFrame values (O, CType, PType, P, Val)
    on conflict (ObjectId , CounterType , PeriodType , Period) do
        update set CounterValue = TimeFrame.CounterValue + Val;
end;
$$ language plpgsql;
```

```
create or replace procedure changeCounter(O object_id , CType int ,
    PType int , P timestamp , Val int)
as $$
declare
    parent object_id;
    storedPeriodTypes int[];
    pnew timestamp;
    pt int;
begin
    parent := (select ParentId from Object where ObjectId = O);
    select StoredPeriods from Counter where CounterType = CType
        into storedPeriodTypes;
```



```

foreach pt in array storedPeriodTypes
    loop
        if pt >= PType then
            select changePtoPT(pt, P) into pnew;
            call changeConcreteCounter(O, CType, pt, pnew, val
            );
        end if;
    end loop;

    if parent is not null then
        call changeCounter(parent, CType, PType, P, val);
    end if;
end;
$$ language plpgsql;

```

#### Листинг А.2 – Объявление общей структуры крона

```

template<typename T>
struct object_finish_t {
    T object_to_process;
    int time_to_process = 0;
};

template<typename T1, typename T2>
struct cron : cron_interface {
    struct heap_element_t {
        object_finish_t<T1> object;
        int* position_ptr;
    };

    struct object_info_element_t {
        T2 data;
        int position = -1;
    };

    std::vector<heap_element_t> check_queue;
    std::unordered_map<T1, object_info_element_t> object_info;
};

```

#### Листинг А.3 – Отрывок скрипта для сравнения реализаций, строящий иерархии

```

$objects = [];
$layers      = [1, 10, 100];
$totalBatches = 100;

```

```

$multiplier = 5;

for ($layer = 0; $layer < count($layers); ++$layer) {
    for ($batch = 0; $batch < $totalBatches; ++$batch) {
        $queries = [];
        for ($j = 0; $j < $layers[$layer] * $multiplier; ++$j) {
            $currentId = $j * $totalBatches + $batch;
            $object = [
                "object_id_full" => [
                    "_" => "objectIdFull",
                    "object_type" => $layer,
                    "object_id" => [$currentId]
                ]
            ];
            if ($layer > 0) {
                $parentId = ($j * $totalBatches + $batch) % ($layers[
                    $layer - 1] * $totalBatches);
                $object["parent_object_id_full"] = [
                    "_" => "objectIdFull",
                    "object_type" => $layer - 1,
                    "object_id" => [$parentId]
                ];
            }
            if ($layer == count($layers) - 1) {
                $objects[] = $object;
            }
            $PG->createObject($object);
            $queries[$j] = $TreeStats->makeQueryInteractObject($object,
                "tree_stats.createObject");
        }
        $results = $TreeStats->callMany($queries);
    }
}

```

# **ПРИЛОЖЕНИЕ Б. ТАБЛИЦЫ**

Таблица Б.1 – Сравнение реализаций

$L_{count}$	$L_i$	$R_{batch}$	PostgreSQL ( $T_p$ )	tree-stats ( $T_t$ )	$\frac{T_p}{T_t}$
2	[100, 10000]	1	1374 мс	331 мс	4.1
		50	17976 мс	1608 мс	11.2
	[10, 100000]	1	1413 мс	331 мс	4.2
		100	30170 мс	2672 мс	11.3
3	[10, 100, 1000]	1	1773 мс	379 мс	4.7
		10	6098 мс	662 мс	6.3
		100	44593 мс	3699 мс	12.1
	[100, 1000, 10000]	1	1806 мс	392 мс	4.6
		100	55129 мс	4141 мс	13.3
4	[10, 100, 1000, 10000]	1	1968 мс	413 мс	4.7
		10	7882 мс	797 мс	9.9
		100	58727 мс	4970 мс	11.8