



Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

ОТЧЁТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

Тип практики: Эксплуатационная практика

Название предприятия: ООО «ВК Цифровые Технологии»

Студент: группа ИУ8-104-2025 л.д. 20У474

Железцов Никита Владимирович

(подпись, дата)

Руководитель от предприятия:

руководитель разработки платформы Tarantool Останевич
Сергей Юрьевич

(подпись, дата)

Руководитель от кафедры:

доцент кафедры ИУ8 Зайцева Анастасия Владленовна

(подпись, дата)

Оценка: _____

РЕФЕРАТ

Отчёт содержит 32 стр., 6 рис., 1 табл., 5 источн.

Данная работа посвящена разработке дизайн-документа по реализации функционала Map-Reduce запроса по репликам в модуле Tarantool-a vshard, обеспечивающего шардирование данных в кластере Tarantool. В ходе работы была проведена аналитика существующего механизма функционирования модуля шардирования, спроектировано решение для добавления вызова `map_callro`, предназначенного для выполнения распределённых операций чтения на всех репликах кластера с учётом требований к согласованности и отказоустойчивости.

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

В настоящем отчете по практике применяются следующие сокращения и обозначения:

БД	—	База данных.
СУБД	—	Система управления базами данных.
DDL	—	Data Definition Language.
DQL	—	Data Query Language.
DML	—	Data Manipulation Language.
DCL	—	Data Control Language.
LSN	—	Log sequence number.
ID	—	Identifier.
UUID	—	Universally unique identifier.

СОДЕРЖАНИЕ

РЕФЕРАТ	3
СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ	4
ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ	6
ВВЕДЕНИЕ	7
ОСНОВНАЯ ЧАСТЬ	8
1 Характеристика организации	8
2 Шардирование	9
2.1 Общая информация	9
2.2 Виды шардирования	9
2.3 Преимущества и недостатки шардирования	11
2.4 Методы шардирования данных	12
3 Обзор шардирования в СУБД Tarantool	14
3.1 Обзор	14
3.2 Архитектура шардированного кластера Tarantool	14
3.3 Маршрутизатор (router)	17
3.4 Ребалансировщик	18
3.5 Миграция бакетов	18
3.6 Системный спейс _bucket	20
3.7 Обработка запроса клиента	20
4 Дизайн-документ по реализации функционала Map по репликам	22
4.1 Мотивация	22
4.2 Текущая работа Map запроса по мастерам (map_callrw)	23
4.3 Дизайн	24
4.3.1 Ref стадия запроса	25
4.3.2 Map запросы и ребалансировка бакетов	25
4.4 Последующие действия	27
4.5 Рассмотренные альтернативы	28
ЗАКЛЮЧЕНИЕ	31
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	32

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

База данных (БД) — это организованная коллекция данных, которая структурирована таким образом, чтобы данные можно было легко хранить, управлять, изменять и извлекать.

Кластер — это совокупность нескольких репликасетов, каждый из которых чаще всего хранит разный набор данных.

Репликасет (replicaset, или шард) — это группа узлов (инстансов), работающих в режиме репликации и объединенных для обеспечения отказоустойчивости и доступности данных.

Система управления базами данных (СУБД) — это программное обеспечение, предназначенное для создания, управления и обеспечения доступа к базам данных. Оно позволяет пользователям определять, создавать, изменять и управлять базой данных, а также обеспечивает взаимодействие между пользователями и базой данных через запросы и команды. Основные функции включают хранение, поиск, обновление и удаление данных, а также обеспечение целостности, безопасности и управления доступом к данным.

Спейс (space) — это основная логическая единица хранения данных Tarantool, аналогичная таблице в традиционных реляционных базах данных. Спейс содержит набор записей, каждая из которых называется кортежем (tuple). Структура спейса определяется схемой, которая включает количество и типы полей в кортежах, а также индексы для быстрого доступа к данным.

LSN — это монотонно возрастающий идентификатор записи.

Vclock — это массив LSN, идентификаторами в котором являются ID узлов. Vclock представляет собой набор логических счетчиков для каждого узла в кластере, позволяя определить, какие изменения были применены на конкретном узле и какие еще предстоит синхронизировать.

ВВЕДЕНИЕ

Место прохождения практики – общество с ограниченной ответственностью «ВК Цифровые Технологии», отдел Research & Development, команда Tarantool Platform Core. Период прохождения практики - с 1 июля 2025 года по 14 июля 2025 года.

В задачи прохождения практики входило:

- а) Разобраться в работе шардирования СУБД Tarantool;
- б) Разработать дизайн-документ по реализации функционала Map-Reduce запроса, обеспечивающего выполнение распределенных операций чтения на всех репликах кластера.

Целью прохождения практики стало проектирование решения для добавления вызова `map_callro`, обеспечивающего выполнение распределённых операций чтения на всех репликах кластера с учётом требований к согласованности и отказоустойчивости. В рамках отчёта представлен обзор функциональности *vshard*, а также детальный дизайн-документ, содержащий архитектурные решения по реализации.

Практическая значимость работы заключается в расширении возможностей модуля *vshard*, что позволит повысить эффективность и безопасность распределённых запросов, улучшить балансировку нагрузки и обеспечить более гибкое использование кластера

ОСНОВНАЯ ЧАСТЬ

1 Характеристика организации

VK Цифровые Технологии – подразделение VK, развивающее продукты и сервисы для цифрового бизнеса. В основе экосистемы решений VK Цифровые технологии лежит многолетний опыт развития интернет-сервисов и технологий на базе открытого кода. VK Цифровые Технологии предоставляет готовые сервисы для решения бизнес задач любой сложности, занимается заказной разработкой и управлением ИТ-инфраструктурой на аутсорсе [1].

В портфеле VK цифровые Технологии — облачные сервисы VK Cloud Solutions, платформа in-memory вычислений Tarantool, платформа взаимодействия бизнеса и государства VK Tax Monitoring, а также линейка программных продуктов для управления персоналом, автоматизации производства и бизнес-процессов.

Tarantool как продукт появился 4 апреля 2016 года, когда Mail.ru Group (на данный момент известная как VK) сообщила о создании нового направления бизнеса, в рамках которого компания начала предоставлять корпоративным клиентам услуги в области хранения данных.

Изначально Tarantool применялся только в собственных проектах Mail.ru, в том числе в почтовом сервисе и облачном хранилище «Облако Mail.Ru». Затем компания превратила эту СУБД в продукт с открытым исходным кодом, который к началу апреля 2016 года внедрен рядом российских и международных компаний. В частности, Tarantool начал использоваться сервисом бесплатных объявлений Avito, социальной сетью знакомств Badoo и разработчиком систем информационной безопасности Wallarm.

На сегодняшний день Tarantool активно используется в банковской сфере (среди клиентов Tarantool можно выделить ВТБ, Альфа Банк, Банк Открытие и Газпромбанк) и для ретейла и e-commerce (Магнит, Wildberries, Ситилинк, X5Group) [2].

2 Шардирование

В данном разделе рассматриваются фундаментальные аспекты шардирования как ключевого метода горизонтального масштабирования баз данных. Исследуются базовые принципы распределения данных, проводится сравнительный анализ вертикального и горизонтального шардирования с практическими примерами их применения. Особое внимание уделяется системному анализу преимуществ и ограничений шардированной архитектуры. Завершает раздел обзор основных методов шардирования с характеристикой их специфических особенностей и областей эффективного применения.

2.1 Общая информация

Шардирование — это подход к организации баз данных, при котором массив данных делится на части и распределяется по различным наборам реплик (шардам). Шард является самостоятельным компонентом кластера и для повышения надёжности часто включает не только мастера, но и реплики — серверы, хранящие копию данных мастера.

Данная технология становится востребованной, когда система исчерпывает возможности вертикального масштабирования (усиления мощности одного сервера) и для дальнейшего роста ей требуется горизонтальное масштабирование (добавление новых машин).

Популярные онлайн-сервисы со временем неизбежно нуждаются в масштабировании для увеличения пропускной способности и быстродействия. Когда, например, социальная сеть разрастается до многомиллионной аудитории, её работа на одном сервере становится невозможной. Для безопасного и целостного распределения информации между несколькими серверами применяется шардированная база данных.

2.2 Виды шардирования

Выделяют два вида шардирования:

- Вертикальное — это вид шардирования, при котором данные распределяются по столбцам. Каждый шард в этой схеме хранит определенный набор колонок со всеми соответствующими строками. Данный подход особенно эффективен в сценариях, когда отдельные столбцы запрашиваются значительно чаще остальных.

В качестве иллюстрации можно рассмотреть базу данных интернет-магазина, содержащую обширную информацию о товарах и клиентах. Для повышения производительности её можно разделить на два шарда: первый будет специализироваться на данных о покупателях, а второй — на информации о продуктах. Это позволит системе загружать только те столбцы, которые необходимы для выполнения конкретного запроса, оптимизируя таким образом использование ресурсов. Пример архитектуры вертикального шардирования представлен на рисунке 1.

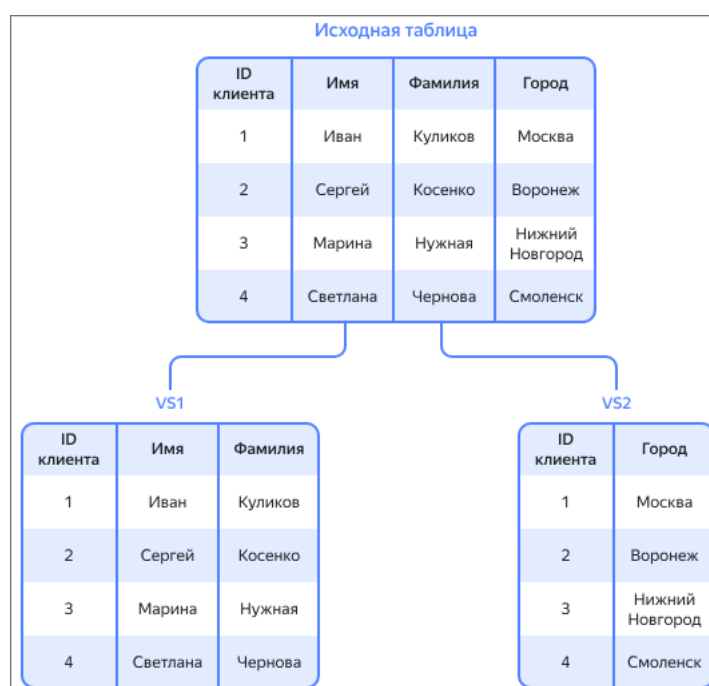


Рисунок 1 – Вертикальное шардирование

- Горизонтальное — это вид шардирования, основанный на разделении строк таблицы в соответствии с определёнными правилами или ключами. При таком подходе каждый шард обладает идентичной схемой столбцов, но содержит уникальные наборы записей. Данная технология обеспечивает эффективное распределение операций чтения и записи между несколькими независимыми серверами, повышая общую производительность и отказоустойчивость системы.

В качестве практического примера можно рассмотреть базу пользователей социальной сети. Для снижения нагрузки на инфраструктуру разработчики применяют горизонтальное партиционирование, используя хеш-функцию от первичного ключа (например, ID пользователя) для опреде-

ления целевого шарда для каждой новой записи. Этот подход гарантирует равномерное распределение данных и запросов по всем узлам кластера. Пример реализации горизонтального шардирования демонстрируется на рисунке 2.

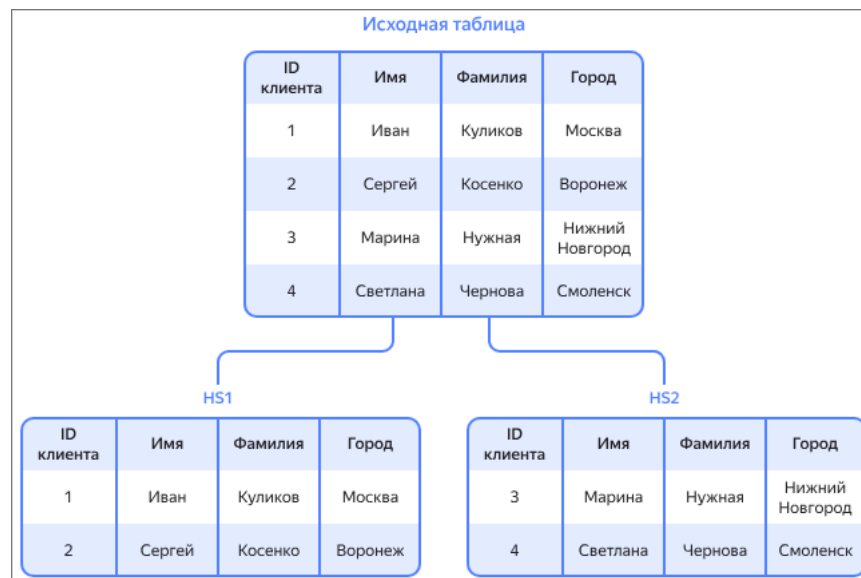


Рисунок 2 – Горизонтальное шардирование

2.3 Преимущества и недостатки шардирования

Когда монолитная база данных достигает пределов вертикального масштабирования, шардирование становится эффективным решением для дальнейшего роста. Данная архитектура предоставляет следующие преимущества:

- **Преодоление аппаратных ограничений.** Популярные сервисы часто упираются в физические лимиты оборудования. Шардирование позволяет распределить информацию среди нескольких серверов, обеспечивая тем самым горизонтальное масштабирование и возможность практически неограниченного расширения.
- **Повышение отказоустойчивости.** Поскольку шарды физически размещаются на разных серверах, выход из строя одного из них не приводит к полному отказу системы. Это означает, что вместо полной недоступности сервиса может перестать функционировать лишь его некоторая часть, что значительно повышает надежность.

- **Увеличение производительности.** Монолитные базы данных часто страдают от конкуренции запросов, что снижает общую скорость работы. Шардирование распределяет нагрузку между узлами, что позволяет увеличить пропускную способность системы и ускорить обработку данных.

Несмотря на свою мощь, шардированная архитектура имеет ряд существенных недостатков, которые делают её применение целесообразным не во всех ситуациях.

- **Высокая сложность внедрения и сопровождения.** Процесс шардирования критически важен, и ошибки на этапе проектирования могут привести к потере или повреждению данных. Кроме того, разработка усложняется, так как исчезает единая точка входа для работы с данными, и команде приходится оперировать множеством изолированных сегментов.
- **Риск несбалансированной нагрузки.** Распределение данных может оказаться неравномерным, если критерии шардирования выбраны неудачно. Это приводит к ситуации, когда один шард (например, с данными популярных пользователей) обрабатывает большую часть запросов, создавая «горячую точку». Для устранения такой несбалансированности часто требуется трудоемкая процедура рещардинга, часто сопровождающаяся простоем.
- **Снижение эффективности сложных запросов.** Операции, которые требуют агрегации данных из нескольких шардов (например, JOIN'ы), выполняются значительно медленнее из-за необходимости сетевого взаимодействия с разными серверами и последующего объединения результатов. Это делает такие запросы более ресурсоемкими по сравнению с выполнением в рамках единой базы данных.

2.4 Методы шардирования данных

В практике распределённых баз данных применяются различные подходы к шардированию, каждый из которых обладает специфическими характеристиками. Выбор оптимальной стратегии зависит от конкретных требований и архитектурных особенностей системы. Наиболее распространённые методы включают:

- **Хеш-шардирование** — распределение данных между шардами осуществляется на основе вычисленного хеш-значения от ключа записи. Данный метод обеспечивает равномерное распределение нагрузки и высокую доступность системы, но усложняет выполнение запросов по диапазонам значений.
- **Диапазонное шардирование** — разделение данных производится согласно заданным интервалам значений ключевых атрибутов. Подход отличается простотой реализации и эффективностью при выполнении запросов в пределах определённого диапазона, однако может приводить к неравномерному распределению нагрузки между шардами.
- **Круговое шардирование** — шарды организуются в логическое кольцо, где каждый узел отвечает за определенный сегмент данных. Запросы маршрутизируются в соответствии с позицией шарда в кольцевой структуре. Метод обеспечивает равномерное распределение запросов, но требует сложной процедуры перераспределения данных при изменении количества шардов.
- **Динамическое шардирование** — система автоматически адаптирует структуру хранения в зависимости от текущей нагрузки и объема данных. Несмотря на высокую гибкость и масштабируемость, данный подход требует сложной системы мониторинга, балансировки нагрузки и тщательно продуманной архитектуры.

3 Обзор шардирования в СУБД Tarantool

В этом разделе детально исследуется реализация механизма шардирования в СУБД Tarantool. Кроме того, анализируются ключевые компоненты модуля шардирования, понимание которых необходимо для реализации предложенного подхода к выполнению Map запросов на репликах.

3.1 Обзор

В Tarantool логически связанные узлы, работающие с идентичными копиями базы данных, объединяются в репликасеты (replicaset). В рамках каждого набора узлов назначаются роли: мастер (master) или реплика (replica). Обработка запросов на изменение данных (DDL, DML, DCL) осуществляется исключительно на мастере, тогда как реплики обслуживают только запросы на чтение (DQL). Между узлами внутри репликасета поддерживается процесс синхронизации данных посредством репликации.

Реализация шардирования в Tarantool обеспечивается модулем *vshard* [3], разработанным на языке Lua. Этот выбор обусловлен тем, что Tarantool сочетает функциональность СУБД с возможностью исполнения прикладной логики, которая реализуется на Lua.

Модуль *vshard* в Tarantool обеспечивает распределение кортежей набора данных среди нескольких шардов, представляющих собой репликасеты. Каждый шард обрабатывает только определённое подмножество общих данных, что позволяет масштабировать систему под повышенную нагрузку путём добавления новых шардов.

Модуль *vshard* предоставляет API маршрутизатора (router) и API хранилища (storage) для разработки приложений, поддерживающих шардирование.

3.2 Архитектура шардированного кластера Tarantool

Рассмотрим распределенный кластер Tarantool, состоящий из подкластеров, называемых шардами, каждый из которых хранит некоторую часть данных. Каждый шард, в свою очередь, представляет собой набор реплик (replica set), состоящий из нескольких реплик, одна из которых служит главным узлом (master node), обрабатывающим все запросы на чтение и запись.

Весь набор данных логически разделен на заранее определенное количество виртуальных бакетов (далее - бакеты), каждому из которых присвоен уникальный номер в диапазоне от 1 до N , где N - общее количество бакетов. Количество бакетов специально выбирается на несколько порядков больше потенциального количества узлов кластера, даже с учетом будущего масштабирования кластера. Например, при M проектируемых узлах набор данных может быть разделен на $100M$ или даже $1000M$ бакетов. При выборе количества бакетов следует соблюдать осторожность: если оно слишком велико, это может потребовать дополнительной памяти для хранения информации о маршрутизации; если слишком мало, это может уменьшить детализацию перебалансировки.

Каждый шард хранит уникальное подмножество бакетов, что означает, что бакет не может принадлежать нескольким шардам одновременно, как показано на рис 3.

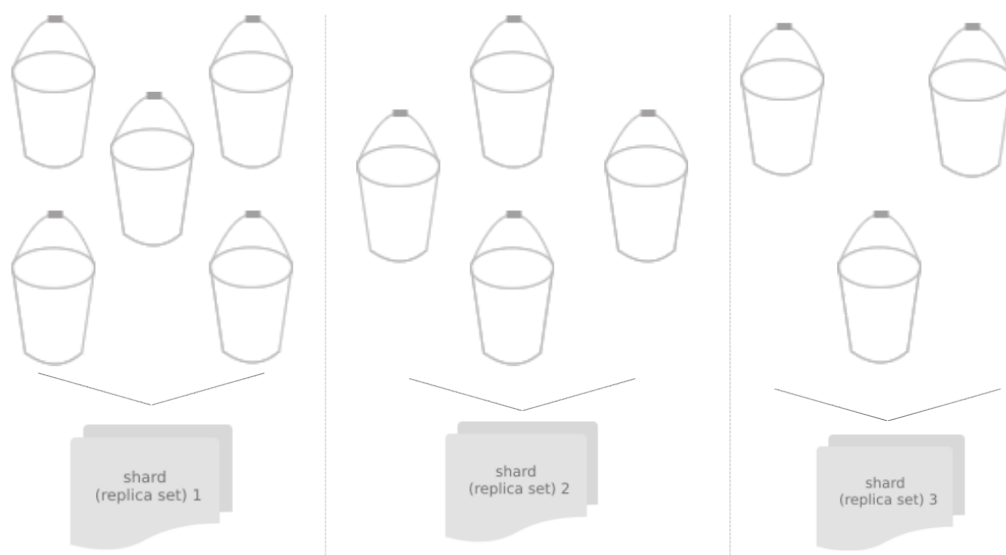


Рисунок 3 – Виртуальные бакеты [4]

Данное соответствие шардов и бакетов хранится в таблице в одном из системных спейсов Tarantool, причем каждый шард содержит только определенную часть этой карты, которая покрывает те бакеты, которые были назначены данному шарду.

Помимо таблицы соответствия, идентификатор бакета также хранится в специальном поле каждого кортежа каждой таблицы, участвующей в шардировании.

Когда шард получает любой запрос (кроме SELECT) от приложения, этот шард проверяет идентификатор бакета, указанный в запросе, по таблице идентификаторов бакетов, принадлежащих данному узлу. Если указанный идентификатор бакета недействителен, запрос завершается со следующей ошибкой: "wrong bucket". В противном случае запрос выполняется, и всем данным, созданным в процессе, присваивается идентификатор бакета, указанный в запросе. Важно отметить, что запрос должен изменять только те данные, которые имеют тот же идентификатор бакета, что и сам запрос.

Хранение идентификаторов бакетов как в самих данных, так и в таблице соответствия обеспечивает целостность данных независимо от логики приложения и делает перебалансировку прозрачной для приложения. Хранение таблицы соответствия в системном спейсе гарантирует согласованность шардирования в случае отказа, поскольку все реплики в шарде разделяют общее состояние таблицы.

Шардированный кластер в Tarantool состоит из:

- **Одного или нескольких наборов реплик:**
 - Каждый набор реплик должен содержать как минимум два экземпляра хранилища (storage)
 - Для обеспечения избыточности рекомендуется иметь 3 или более экземпляров хранилища в наборе реплик
- **Одного или нескольких маршрутизаторов (router):**
 - Количество экземпляров маршрутизаторов не ограничено
 - Его следует увеличивать, если существующие экземпляры маршрутизаторов становятся узким местом из-за нагрузки на ЦПУ или ввод-вывод
- **Балансировщика (Rebalancer)**

Структура шардированного кластера представлена на рис 4

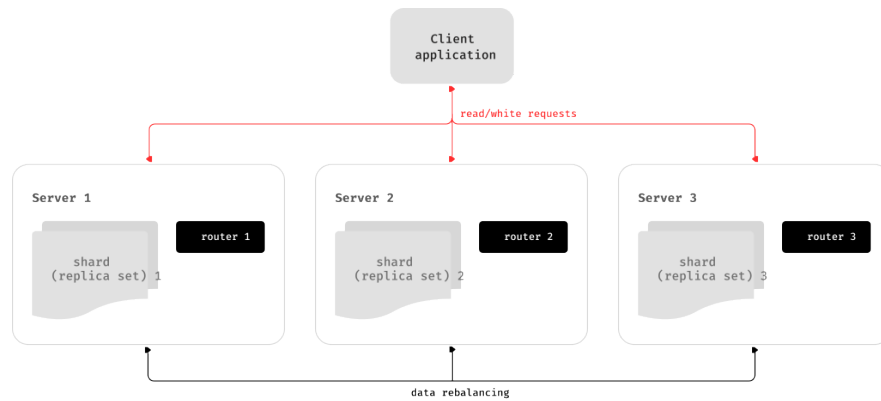


Рисунок 4 – Архитектура шардированного кластера [4]

3.3 Маршрутизатор (router)

Маршрутизатор (далее роутер) представляет собой автономный программный компонент, который направляет запросы на чтение и запись от клиентского приложения к соответствующим шардам.

Все запросы от приложения поступают в шардированный кластер через роутер. Роутер обеспечивает прозрачность топологии шардированного кластера для приложения, скрывая от него:

- количество и расположение шардов,
- процесс перебалансировки данных,
- факт и процесс отработки отказа при сбое реплики.

Роутер может самостоятельно вычислять идентификатор бакета при условии, что приложение явно определяет правила вычисления идентификатора бакета на основе данных запроса. Для этого роутер должен знать схему данных.

Роутер не имеет постоянного состояния и не хранит топологию кластера, а также не выполняет балансировку данных. Это автономный программный компонент, который может работать на уровне хранилища или на уровне приложения в зависимости от особенностей приложения.

Роутер поддерживает постоянный пул соединений со всеми хранилищами, создаваемый при запуске. Такой подход помогает избежать ошибок конфигурации. После создания пула роутер кэширует текущее состояние спейса `_bucket` для ускорения маршрутизации. В случае перемещения идентификатора бакета на другое хранилище в результате ребалансировки данных или при переходе одного из шардов на реплику, роутер обновляет таблицу маршрутизации.

Шардирование не интегрировано в какую-либо централизованную систему хранения конфигурации. Предполагается, что само приложение обрабатывает все взаимодействия с такими системами и передает параметры шардирования. При этом конфигурация может изменяться динамически - например, при добавлении или удалении одного или нескольких шардов:

- Для добавления нового шарда в кластер системный администратор сначала изменяет конфигурацию всех роутеров, а затем конфигурацию всех хранилищ.
- Новый шард становится доступным уровню хранилища для ребалансировки.
- В результате ребалансировки один из виртуальных бакетов перемещается на новый шард.
- При попытке доступа к виртуальному бакету роутер получает специальный код ошибки, указывающий новое местоположение бакета.

3.4 Ребалансировщик

Ралансировщик представляет собой фоновый процесс ребалансировки, который обеспечивает равномерное распределение бакетов между шардами. В процессе ребалансировки происходит миграция бакетов между наборами реплик.

Балансировщик периодически "просыпается" и перераспределяет данные из наиболее загруженных узлов в менее загруженные узлы. Ребалансировка запускается, если дисбаланс набора реплик превышает заданный в конфигурации пороговый уровень.

Дисбаланс набора реплик рассчитывается следующим образом:

$$|etalon_bucket_number - real_bucket_number| / etalon_bucket_number * 100 \quad (1)$$

3.5 Миграция бакетов

Набор реплик, с которого выполняется миграция бакета, называется источником (source); целевой набор реплик, на который выполняется миграция, называется приемником (destination).

Во время миграции бакет может находиться в различных состояниях:

- **ACTIVE** – бакет доступен для запросов на чтение и запись.

- **PINNED** – бакет заблокирован для миграции на другой набор реплик. В остальном заблокированные бакеты аналогичны бакетам в состоянии **ACTIVE**.
- **SENDING** – бакет в настоящее время копируется на набор реплик приемник; запросы на чтение к набору реплик источник все еще обрабатываются.
- **RECEIVING** – бакет в настоящее время заполняется; все запросы к нему отклоняются.
- **SENT** – бакет был мигрирован на набор реплик приемник. Роутер использует состояние **SENT** для вычисления нового местоположения бакета. Бакет в состоянии **SENT** автоматически переходит в состояние **GARBAGE** через 0.5 секунды.
- **GARBAGE** – бакет уже был мигрирован на набор реплик приемник во время перебалансировки; или бакет изначально находился в состоянии **RECEIVING**, но во время миграции произошла ошибка. Бакеты в состоянии **GARBAGE** удаляются сборщиком мусора.

Процесс миграции бакетов представлен на рис 5. Она выполняется следующим образом:

- На целевом наборе реплик (destination) создается новый бакет, которому присваивается состояние **RECEIVING**, начинается копирование данных, и бакет отклоняет все запросы.
- Бакету в исходном наборе реплик (source) присваивается состояние **SENDING**, и бакет продолжает обрабатывать запросы на чтение.
- После завершения копирования данных бакет в исходном наборе реплик переводится в состояние **SENT** и начинает отклонять все запросы.
- Бакет в целевом наборе реплик переводится в состояние **ACTIVE** и начинает принимать все запросы.

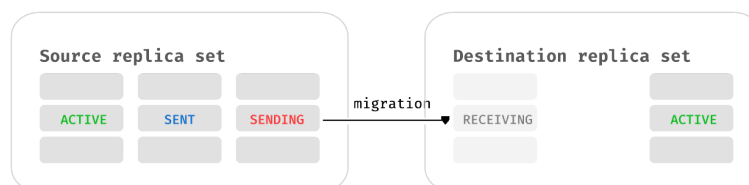


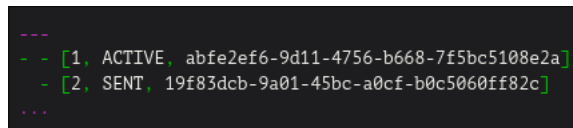
Рисунок 5 – Пересылка бакета [4]

3.6 Системный спейс `_bucket`

В системном спейсе `_bucket` каждого набора реплик хранятся идентификаторы бакетов, присутствующих в данном наборе реплик. Спейс содержит следующие поля:

- `bucket` – идентификатор бакета
- `status` – состояние бакета
- `destination` – UUID целевого набора реплик

Пример вывода команды `_bucket:select{}` представлен на рис. 6.



```
---
- - [1, ACTIVE, abfe2ef6-9d11-4756-b668-7f5bc5108e2a]
- - [2, SENT, 19f83dcb-9a01-45bc-a0cf-b0c5060ff82c]
...
```

Рисунок 6 – Содержание спейса `_bucket`

После завершения миграции бакета в таблице заполняется поле UUID целевого набора реплик. Пока бакет находится на исходном наборе реплик, значение UUID целевого набора реплик равно `NULL`.

3.7 Обработка запроса клиента

Запросы к базе данных могут выполняться приложением или с использованием хранимых процедур. В любом случае, идентификатор бакета должен быть явно указан в запросе.

Все запросы сначала перенаправляются роутеру. Единственной операцией, поддерживаемой роутером, является `call`. Данная операция выполняется с помощью функции `vshard.router.call()`:

```
result = vshard.router.call(<bucket_id>, <mode>, <function_name>, \\  
    {<argument_list>}, {<opts>})
```

Обработка запросов выполняется следующим образом:

- а) Роутер использует идентификатор бакета для поиска соответствующего набора реплик в таблице маршрутизации.
- б) Если соответствие идентификатора бакета набору реплик неизвестно роутеру (`fiber` обнаружения еще не заполнил таблицу), роутер отправляет запросы ко всем хранилищам, чтобы определить местоположение бакета.
- в) После обнаружения бакета шард проверяет:

- наличие бакета в системном спейсе `_bucket` набора реплик;
 - находится ли бакет в состоянии `ACTIVE` или `PINNED` (для запросов на чтение также допустимо состояние `SENDING`).
- г) Если все проверки успешны, запрос выполняется. В противном случае он завершается с ошибкой: `"WRONG_BUCKET"`.

4 Дизайн-документ по реализации функционала Map по репликам

- Текущая реализация Map запроса по репликам у клиента ведет к дублированию, потере и повреждению данных;
- В vshard будут добавлены новые запросы: `map_callro`, `map_callre`, `map_callbro`, `map_callbre`;
- `map_*` запросы (даже те, которые выполняются на репликах) замедляют ребалансировку. Пользователь может регулировать распределение между map запросами и перевозом бакетов с использованием опций `sched_ref_quota`, `sched_move_quota`.

4.1 Мотивация

В бизнес-логике клиента очень много map-запросов, однако vshard на данный момент не предлагает никакого решения для совершения map-запросов по репликам, вследствие чего клиент вынужден писать собственные решения с использованием `vshard.router.routeall`, которые выполняются с рисками нарушения консистентности и целостности получаемых данных (дублирование и потеря).

Базово сейчас `map_callro` у клиента выглядит так (еще есть логика на пропуск репликасетов из `routeall` при определенных условиях):

```
for _, rs in pairs(vshard.router.routeall()) do
  rs:callbro('box.space.card:select()', nil, {timeout = <number>})
end
```

Это крайне небезопасно:

- а) **Дублирование данных** - бакет может находиться в процессе пересылки (`state = SENDING`) или уже даже быть послан (`state = SENT`). Данные не удаляются во время пересылки, только после ее полного окончания, поэтому когда бакет переезжает данные есть на двух шардах. При `select` по спейсу получим дублирование данных. Даже если бакет полностью переехал, дублирование возможно: пришли на первый репликасет, там

данные еще есть, пока идем до второго, они переехали, пришли на второй, там эти же данные уже есть. Если бакет в процессе удаления, то можно прочитать часть бакета, а полностью он будет только там, куда уехал, снова дублирование.

- б) **Потеря данных** - мы идем на первый репликасет, данных там пока нет, нам вернули результат. Пока идем до следующего, он успевает переслать бакет на первый репликасет (полностью переслать, с удалением данных). Мы дошли до него, данных там тоже не будет. Мар-запрос потерял данные.
- в) **"Повреждение данных"** - можно прочитать неконсистентное состояние данных (`state = GARBAGE`). Допустим пользователь добавлял в один бакет в одной транзакции таплы 1, 2, 3, 4, 5. Бакет уехал. Таплы 1, 2, 3 уже удалились. Таплы 4 и 5 еще нет. Запрос их прочитает, т.е. половину пользовательской транзакции.

Vshard-у следует предоставить собственное решение для выполнения мар-запроса по репликам, который не будет дублировать и терять бакеты.

4.2 Текущая работа Map запроса по мастерам (`map_callrw`)

`vshard.router.map_callrw` принимает в качестве аргумента функцию, которая вызывается на мастерах в кластере с гарантией, что в случае успешного выполнения она была выполнена при доступности всех бакетов для чтения и записи. Под консистентностью в рамках Map-Reduce понимается, что все данные были доступны и не перемещались в процессе выполнения запросов Map. Чтобы сохранить консистентность, добавлена третья стадия - Ref. Таким образом, алгоритм на самом деле называется Ref-Map-Reduce.

Ref-ы отправляются до этапа Map, чтобы закрепить бакеты на репликасетах и гарантировать, что они не будут перемещаться до завершения стадии Map. Если хотя бы один Ref не удалось проставить, то пользователю возвращается ошибка.

Если все Ref-ы успешно завершены, начинается рассылка запросов Map. Эти запросы выполняют пользовательскую функцию и после удаляют Ref-ы, чтобы снова разрешить ребалансировку. Рассылка запросов Map производится параллельно, ко всем необходимым мастерам.

На уровне стораджей существуют дополнительные механизмы, чтобы гарантировать, что Map-Reduce не блокирует ребалансировку навсегда и наоборот: что ребалансировка не заблокирует полностью Map-Reduce. Это конкурирующие процессы. В vshard есть опции: `sched_ref_quota`, `sched_move_quota`. Если `move_quota = 2` и `ref_quota = 15`, то это означает, что максимум 2 бакета могут переехать, если есть запросы на ref. И наоборот, максимум 15 рефов может быть сделано, если нужно перевозить бакет.

Подробнее про работу `map_callrw` можно прочитать в дизайн-документе соответствующем дизайн-документе [5].

Аргументы `vshard.router.map_callrw`:

- **func**: Имя вызываемой функции.
- **args**: Аргументы функции, переданные в формате netbox (в виде массива).
- **opts.timeout**: Таймаут в секундах. Учтите, что Ref-ы могут оставаться на хранилищах в течение всего этого таймаута, если что-то пойдет не так, например, возникнут сетевые проблемы. Поэтому лучше не использовать значение больше, чем необходимо.
- **opts.return_raw**: true/false. Если указано, возвращаемые значения не декодируются в нативные объекты Lua и остаются упакованными как объект msgpack (см. модуль msgpack). По умолчанию все значения декодируются. Это может быть нежелательно, если возвращаемые значения будут сразу пересылаться по сети.
- **opts.bucket_ids**: Массив ID бакетов, которые должны быть охвачены Map-Reduce. Если она указана, Map-Reduce выполняется только на мастерах, содержащих хотя бы один из этих бакетов. В противном случае Map-Reduce выполняется на всех мастерах кластера.

4.3 Дизайн

В map-запросах все фазы Ref-Map выполняются на одной и той же реплике:

- **map_callrw** – выполняется только на мастере. Реализован в текущей версии.

- `map_callro` – реплика выбирается на основе весов в конфигурации роутера (чем меньше вес, тем реплика приоритетнее). Такой репликой может являться и мастер (при использовании весов по умолчанию мастер является наиболее приоритетной репликой). Выбор реплики не является постоянным и может временно меняться, если самая приоритетная реплика недоступна.
- `map_callre` – предпочтение отдается не мастеру (но все еще может быть выполнена на мастере как последняя мера, если все реплики не ответили).
- `map_callbro` – round-robin балансировка по всем экземплярам в репликасете, может быть как мастер, так и реплика.
- `map_callbre` – round-robin балансировка, но с предпочтением пропускать мастера.

4.3.1 Ref стадия запроса

Для `map` запросов по репликам применяются те же рефы, что используются сейчас для мастеров, `ref` полностью локальный (если экземпляр не мастер, то запрос на мастера не совершается). Реф происходит до запроса ко всем репликам, в отдельную фазу, как и для `map_callrw`. Ref так же как и в случае с `map_callrw` дожидается, чтобы все бакеты на экземпляре были **ACTIVE** или **PINNED**. Если не получается зарефать (идет ребалансировка), возвращается ошибка, ни один `map` запрос не будет выполнен. Если `ref` успешно создан на всех репликах, выполняется пользовательская функция, после чего реф удаляется.

Ref блокирует все бакеты на экземпляре, пока висит Ref, ребалансировка невозможна.

4.3.2 Map запросы и ребалансировка бакетов

Мастеру приходит запрос на перевоз `N` бакетов. Он собирает эти `N` бакетов и перед началом отправления делает `sched.move_start` локально. В случае успеха мастер переводит бакеты в состояние **SENDING**. Далее, данные не отправляются/получаются, пока мы не получим на это разрешение всех реплик в нашем репликасете. Мастер дожидается на каждой из реплик выполнения функции `bucket_move_prepare_replica(timeout, vclock_of_first_bucket)`, где `vclock` – это `vclock` мастера после того, как был изменен статус первого измененного бакета. Эта функция выполняется на реплике следующим образом:

- Вызывает `sched.move_start` чтобы создать конкуренцию за ресурсы на репликах и обеспечить заданное пользователем (с помощью `sched*_quota` опций) распределение между перевозом бакетов и ref запросами.
- Дождется, чтобы реплика достигла необходимого `vclock`. Т.е. получила хотя бы один **SENDING** бакет. Этого достаточно, чтобы новые рефы не могли быть созданы и ждать, пока отреплицируются все, смысла ждать нет.
- Вызывает `sched.move_end(1)`. Отныне новые рефы не могут быть созданы, так как бакет перемещается.

Если хотя бы одна реплика ответила ошибкой, бакеты не будут посылаться, ошибка отправления бакета. Не меняем состояние бакета, в случае ошибок этим занимается `recover` (не забыть разбудить в случае фейла) и `gc`. Если все реплики смогли подтвердить перевоз бакетов, то обращаемся к мастеру, на который данные будут отправляться: `bucket_recv_prepare_master(timeout, buckets)`. Он переводит бакеты в состояние **RECEIVING**, делает вызов по репликам `bucket_move_prepare_replica`, чтобы убедиться, что новые рефы не смогут быть созданы и дождаться окончания `map_callro`. Если успешно, возвращает `ok`.

Вся работа мастера, описанная выше объединяется в функцию `bucket_send_prepare_master(timeout, buckets)`.

Таким образом на посылку/получение одного батча (откуда посылает: батч - все `route`, которые пришли, куда посылает: батч - набор бакетов от конкретного репликасета) нужно будет совершить только по одному запросу на каждую из реплик. Дальше при пересылке бакетов работают воркеры, как обычно (но не переводя в состояние **SENDING** и то, что описано выше).

Чтобы предотвратить одновременную посылку одного и того же бакета (например, когда ребалансер пытается послать и пользователь вызывает `bucket_send`), `M.rebalancer_transferring_buckets[bid] = 'preparing'` выполняется для каждого из бакетов перед локальным `sched.move_start` на мастере (в самом начале `bucket_send_prepare_master`). Затем когда воркер начинает работу он проверяет, что статус бакета **preparing** (можно было бы добавить статус **prepared** после того, как дождались разрешения реплик, но выглядит бессмысленно, мы не должны отдавать работу в воркеры, если бакет не разрешили пересылать), меняет его в **transferring**. На принимающей сто-

роне состояние бакета меняется на `preparing` в `bucket_recv_prepare_master` (recovery должен раззуливать эти статусы и подчищать их, если на отправляющей стороне `rebalancer_transferring_buckets` в `nil`). На стадии `bucket_recv` состояние бакета меняется на `transferring`. По умолчанию `M.rebalancer_transferring_buckets[bid] = nil`.

Если `bucket_recv` видит, что бакета не существует, тогда он вызывает `bucket_move_prepare_replica`. Необходимо, чтобы ребалансинг работал, если отправитель на старой версии, а получатель на новой. Если отправитель на новой версии, то пересылка бакета будет падать на стадии `bucket_recv_prepare_master`.

4.4 Последующие действия

Описанное в данном пункте будет реализовано только по запросу от пользователей. В изначальную имплементацию `map` запросов этот пункт не входит.

Во все `map_*` запросы добавляется новая опция `consistency_coverage_mode`. При обоих режимах выполняется попытка остановить ребалансировку!

- `consistency_coverage_mode = 'full'`. Запрос будет выполнен везде или не будет выполнен вовсе, запрос гарантированно будет без дублирования данных, без их потери, т.е. будет выполнен на каждом бакете и ровно один раз.
`map_*` запрос выполняется в две фазы: `Ref` - попытка приостановить ребалансировку на всех инстансах, `Map` - выполнение пользовательской функции. Необходим в первую очередь для `map_callrw`. Если ошибка на стадии `Ref` - возврат ошибки пользователю, функция не будет исполнена нигде.
- `consistency_coverage_mode = 'accidental'`. Запрос будет выполнен там, где удастся остановить ребалансировку. Там, где он выполнится, дублирования данных не будет, т.е. функция не будет исполнена дважды на одном и том же бакете, "not more than once". Но данные в рамках кластера могут быть частично прочитаны и записаны, нет гарантии "at least once" для каждого из бакетов.

`map_*` запрос тоже выполняется в две фазы. Единственное отличие от `full` режима: если на стадии `Ref` не удалось выполниться на всех инстансах, то стадия `Map` выполняется на тех инстансах, где `Ref` прошел успешно. Всегда возвращается таблица типа `{<replicaset_uuid> = <returned_value>, ...}`.

Полезен при `map_*` запросах на чтение.

Первый режим необходим для работы `map_callrw`. Необходимо гарантировать, что запрос выполнен на всех репликасетах, а не на их части. Ибо если мы записали что-то только на части репликасетов, то это беда, частичное обновление данных. Однако, если пользователь, например, читает только с мастеров (например, для обеспечения максимальной согласованности данных), то имеет смысл использовать `accidental`.

Второй режим будет очень полезен для `map_callro`, так как в большинстве случаев лучше прочитывать хоть что-то (например делаем `select` по спейсу), чем не читать вообще ничего. Однако если данные невозможно использовать частично, `consistency_coverage_mode` все равно должен стоять в `full`.

Режим `full` существует уже сейчас в `map_callrw`. Будет добавлен только `accidental`. Для `map` запросов по репликам в изначальной имплементации реализовываем только `full` режим, `accidental` добавляется в тикет.

4.5 Рассмотренные альтернативы

Вызывать `sched.move_start` на репликах перед изменением состояния бакета

Была идея, чтобы перед отправлением/получением бакета, до изменения состояния бакета совершать `sched.move_start`. Только если сумели дождаться 0 рефов везде, можно начинать ребалансировку, производится перевод бакета в нужное состояние на мастере, бакет перевозится. Реплика автоматически зовет `sched.move_end` в `on_replace` триггере когда получает по репликации `replace` в `_bucket`. Если же где-то есть рефы и мы не смогли дождаться, то отменяем все с помощью `sched.move_end` с мастера.

Однако тут мы наблюдаем стандартную проблему 2PC. Если мы не используем таймаут, после которого автоматически зовется `sched.move_end`, то, если мастер падает сразу после вызова `sched.move_start`, реплика навсегда заблокирована. Если мы используем таймауты, то можем получить неконсистент-

ные данные, так как таймаут может отработать раньше, чем придет replace в `_bucket`, будет создан ref для `map_call`, а потом к нам долетит replace: во время исполнения функции пользователя мы нарушили гарантии того, что состояние бакетов не будет меняться.

При посылке бакета разрешать рефать реплику

Была идея разрешить рефать реплики, пока бакет находится в состоянии **SENDING**. Появляется новый тип глобального рефа (определенного в модуле `ref.lua`): `ref.add_ro`, старый переименовывается в `ref.add_rw`. В отличие от текущего `rw` ref-а он ждет, чтобы все бакеты были читаемыми (т.е. **ACTIVE**, **PINNED**, или **SENDING**). Именно этим глобальным рефом и пользуются реплики при `map_call`.

В случае отправления бакета. Переводим бакет в состояние **SENDING**, начинаем отправлять. Читать **SENDING** бакеты можно. Он не будет удален, пока существуют рефы на этот бакет, однако нужно заставить gc учитывать го рефы на стораджах. Поведение такое же, как для обычных `call` запросов: могут существовать рефы, даже если бакет в состоянии **SENT**, **GARBAGE**.

В случае получения бакета (**RECEIVING**) рефать такой сторадж запрещено, мастер дожидается 0 го ref-ов после перевода бакета в состоянии **RECEIVING**.

Тут проблема в том, что бакет может читаться в состоянии **SENT/GARBAGE**. Начали выполнять `map_callro` прямо перед окончанием переезда, успели зарефать на реплике, откуда посылалось (т.е. до перевода из **SENDING** в **SENT**), пришли на тот, куда прислалось, когда там бакет уже в **ACTIVE**, получим дублирование данных.

Включение `map_callro` только по опции в конфигурации

Предполагается, что `map_call` с `mode = 'read'` будет использоваться редко, а потому нет смысла усложнять ребалансировку бакетов для пользователей, которые не пользуются данной функцией. Для этого добавляется новая опция конфигурирования `vshard` (будет проброшена в `cartridge`): `enable_read_map_calls` (name subject to change) - опция может быть задана на уровне репликасета или глобально в конфигурации (для всех репликасетов сразу). Включает проверки собственного репликасета на ref-ы при ребалансировке (пересылке бакетов и recovery). По умолчанию равна `false`.

Когда эта опция `false` на роутере, роутер будет сразу же возвращать ошибку `UNSUPPORTED`, без запросов к репликам, при попытке использовать любой `map_call` с `mode = 'read'`. Если же на роутере она `true`, а на сторадже `false`, то реплика будет возвращать эту ошибку когда роутер делает запрос. Чтобы `map_call` по репликам работал, опция должна быть включена и на роутерах, и на стораджах.

Приняли решение отказаться от этой опции, чтобы всегда обеспечивать максимальные проверки при ребалансинге.

ЗАКЛЮЧЕНИЕ

В ходе прохождения практики был проведён комплексный анализ механизма шардирования в СУБД Tarantool. Основное внимание было уделено изучению архитектуры модуля **vshard**, принципов распределения данных и обеспечения согласованности при выполнении операций в шардированном кластере.

Были рассмотрены и проанализированы существующие подходы к реализации Map-Reduce запросов по репликам. В результате исследования выявлены ключевые проблемы, связанные с обеспечением консистентности данных при выполнении распределённых запросов, и предложена альтернативная реализация.

Практическая значимость работы заключается в:

- Систематизации знаний о работе шардированного кластера Tarantool;
- Выявлении ограничений существующей реализации модуля **vshard**;
- Разработке предложений по расширению функциональности для поддержки Map-Reduce операций по репликам.

Полученные результаты могут быть использованы для дальнейшего развития модуля шардирования Tarantool и улучшения его безопасности. Проведённое исследование демонстрирует важность комплексного подхода к проектированию распределённых систем и необходимость тщательного анализа требований к согласованности данных.

Результаты работы подтверждают возможность реализации эффективного механизма выполнения Map-Reduce запросов по репликам в шардированной среде с соблюдением требований к консистентности данных и производительности системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. VKTech. Официальный сайт VK Tech. — 2025. — URL: <https://tech.mail.ru/> (дата обращения 14.07.2025).
2. Tarantool. Официальный сайт Tarantool. — 2025. — URL: <https://www.tarantool.io/ru/> (дата обращения 14.07.2025).
3. GitHub. Исходный код модуля tarantool/vshard. — 2025. — URL: <https://github.com/tarantool/vshard/> (дата обращения 28.07.2025).
4. Tarantool. Sharding architecture. — 2025. — URL: https://www.tarantool.io/en/doc/latest/platform/sharding/vshard_architecture/ (дата обращения 28.07.2025).
5. Shpilevoy V. Consistent Map-Reduce in vshard. — 2021. — URL: <https://github.com/tarantool/vshard/discussions/261/> (дата обращения 28.07.2025).