Миграция высоконагруженного кластера с Redis на Apache Cassandra: архитектурные решения и анализ производительности

Г.Г. Гаджилов, А.В. Хатунов

Аннотация **– В работе представлен кейс миграции высоконагруженной системы с Redis 6.2.x на Apache Cassandra 4.1.x в конфигурации двух дата-центров (RF=3+3, `LOCAL_QUORUM`). Приведена воспроизводимая методика нагрузочного тестирования (YCSB-A, Zipf, 100 млн ключей, 1 КБ запись, прогрев/замер) и сравнение задержек p95/p99 и пропускной способности с учётом конфигураций операционной системы, файловой структуры, дискового пространства и конфигурации виртуальной JAVA-машины. Показаны результаты испытаний на отказ при потере узла и потере дата-центра. Представлены регламенты эксплуатации, включая восстановление и резервное копирование. Обсуждены варианты компромиссной оптимизации выбора стратегии уплотнения данных с анализом альтернативных подходов, таких как стратегия равномерного уплотнения LCS и стратегия для временных рядов TWCS). Рассмотрено влияние фоновых задач уплотнения и синхронизация данных между узлами в высоконагруженной распределённой системе хранения данных на «хвосты задержек» - временные скачки длительности задержки, ведущие к деградации производительности системы при высоких нагрузках.**

Ключевые слова **—NoSQL, Apache Cassandra, Redis, миграция данных, распределённые системы, производительность, JVM оптимизация, воспроизводимость экспериментов, отказоустойчивость**

Введение

Утверждение же целевых состояний имеет смысл в обратном порядке, начиная с формулировки стратегии и заканчивая целевыми моделями бизнес-процессов.

Рост требований к доступности и горизонтальной масштабируемости распределённой системы хранения данных привёл к необходимости миграции части функциональности с системы управления базами данных в памяти оперативной памяти сервера Redis (REmote DIctionary Service) [1] на распределённую систему управления базами данных Apache Cassandra [2-4]. Целью работы было обеспечение предсказуемости «хвостов задержек» при смешанной нагрузке чтения/записи и устойчивость распределённой системы хранения с архитектурой множественных дата-центров в системе без ручного распределения данных между независимыми частями (шардами) на уровне приложения.

Миграция высоконагруженных распределённых систем хранения данных между различными платформами [5-6] требует строгого планирования, воспроизводимой методики и глубокого анализа архитектурных решений. В данном исследовании описан процесс перехода от Redis 6.2.7 в режиме Cluster OSS к Apache Cassandra 4.1.3 в условиях интенсивной обработки запросов. Цель миграции заключалась в устранении архитектурных ограничений Redis при интенсивных операциях записи, повышении отказоустойчивости и снижении совокупных затрат на инфраструктуру.

Рассмотрен реальный пример миграции высоконагруженной распределенной системы хранения данных с конфигурации Redis 6.2.7 (Cluster mode, OSS) на конфигурацию Apache Cassandra 4.1.3. Миграция выполнялась в условиях обработки порядка ста тысяч запросов в секунду. Исходная инфраструктура включала 18 узлов Redis (4 vCPU, 60 ГБ ОЗУ, 50 ГБ SATA SSD на узел), новая архитектура — 8 узлов Cassandra (16 vCPU, 32 ГБ ОЗУ, 500 ГБ NVMe на узел), распределённых по двум дата-центрам (5+3). Представлены параметры тестирования, профиль нагрузки, методика измерений, конфигурации операционной системы, основного файла конфигурации для кластера Cassandra (cassandra.yaml), параметры виртуальной машины Java (jvm.options), а также результаты нагрузочных тестов и тестов отказоустойчивости с доверительными интервалами. Показано увеличение производительности и эффективности использования ресурсов при сохранении SLA p99 для операций записи < 45 мс. Методика воспроизводима и позволяет повторить эксперимент на аналогичном оборудовании.

I. Архитектурные основы и планирование миграции

I.I Сравнительный анализ систем хранения данных

Система управления базами данных в памяти оперативной памяти сервера Redis (REmote DIctionary Service) представляет собой структуру данных в памяти, функционирующую как база данных, кэш и брокер сообщений. Архитектура Redis основана на однопоточной модели с репликацией ведущий-ведомый, что обеспечивает высокую производительность для операций чтения, но создает существенные ограничения для горизонтального масштабирования в условиях интенсивной записи. В рамках данной работы исходным кластер Redis состоял из 18 узлов с конфигурацией 4 процессорных ядра, 60 ГБ оперативной памяти и 50 ГБ дискового пространства на каждом узле.

Распределённая система управления базами данных Apache Cassandra реализует модель данных с широкими семействами столбцов на основе распределенной архитектуры без единой точки отказа. Система

использует последовательное хэширование для распределения данных и протокол Gossip для поддержания состояния кластера. Ключевым преимуществом является линейное масштабирование производительности при добавлении узлов.

I.II Обоснование выбора Apache Cassandra

Переход на Cassandra обусловлен несколькими критическими факторами, превосходящими возможности Redis в контексте требований к рассматриваемой системе:

Масштабируемость и распределение данных. Redis Cluster ограничен необходимостью ручного шардирования и перебалансировки при изменении топологии кластера. Cassandra автоматически распределяет данные между узлами с использованием последовательного хэширования, обеспечивая равномерную нагрузку без административного вмешательства.

Эффективность использования вычислительных ресурсов. Анализ показал неэффективное использование процессорной мощности в Redis кластере: при конфигурации 4 CPU на узел утилизация процессора составляла менее 40% даже под пиковой нагрузкой из-за однопоточный архитектуры. В отличие от этого, Cassandra с 16 CPU на узел обеспечивает полноценное многопоточное выполнение операций чтения и записи.

Отказоустойчивость и географическое распределение. Redis требует настройки следящего узла (sentinel) для обеспечения высокой доступности и не предоставляет встроенной поддержки репликации между дата-центрами. Напротив, Cassandra обеспечивает многоуровневую отказоустойчивость через топологически осведомленную репликацию и автоматическое восстановление после сбоев.

Модель консистентности. Redis обеспечивает конечную согласованность с возможностью потери данных при отказах. Cassandra предоставляет возможность использовать различные настраиваемые уровни целостности, достоверности и согласованности (консистентности) данных в интервале от «возможной согласованности» до «строгой согласованности», позволяя оптимизировать баланс между производительностью и надежностью данных.

Оптимизация хранения данных. Redis кластер с 60 ГБ ОЗУ на узел требовал постоянного мониторинга использование памяти из-за применения архитектуры «в оперативной памяти». Cassandra с 32 ГБ ОЗУ и 500 ГБ дискового пространства обеспечивает гибридную модель хранения данных с эффективным кэшированием горячих данных.

I.III Планирование топологии кластера

Проектирование новой архитектуры основывалось на анализе паттернов доступа к данным и требованиях к консистентности. Конфигурация с 8 узлами (5+3) в двух дата-центрах обеспечивает:

- Отказоустойчивость на уровне дата-центра при использовании стратегии репликации NetworkTopologyStrategy
- Оптимальное соотношение консистентности и производительности при настройке консистентности LOCAL_QUORUM
- Эффективное использование ресурсов с учетом характеристик рабочей нагрузки

Каждый узел кластера Cassandra оснащен 16 процессорными ядрами, 32 ГБ оперативной памяти и 500 ГБ дискового пространства, что обеспечивает значительное увеличение вычислительной мощности при одновременном сокращении общего количества узлов.

I.IV Оптимизация Java Virtual Machine

Критическим фактором производительности стала оптимизация JVM при использовании 32 ГБ оперативной памяти. Конфигурация памяти обеспечивает оптимальную работу механизма оптимизации памяти в виртуальной машине на принципе «сжатые обычные указатели на объекты» (compressed ordinary object pointers, COOPs) в HotSpot JVM, что приводит к существенному улучшению производительности:

Эффективность использования памяти. При конфигурации размера кучи (Heap Size) менее 32 ГБ виртуальной машины Java автоматически включает механизм COOPs, сокращая размер объектных ссылок с 64 до 32 бит. Это приводит к уменьшению объем памяти на 20-30% и улучшению локализации кэша.

Производительность сборки мусора. Оптимальный размер области памяти JAVA heap позволяет использовать механизм G1 автоматического управления памятью виртуальной машины Java с минимальными длительностями пауз. Конфигурация обеспечивает паузы менее 10ms в 99% случаев, что критично для поддержания низкой задержки ответов.

Пропускная способность. Механизм COOPs увеличивают эффективную пропускную способность памяти за счет уменьшения количества пропусков в кэше и более эффективного использования CPU-кэшей L1/L2/L3.

II. Исходные данные для организации миграции

Исходная система эксплуатировала кластер Redis, оптимизированный под низкие латентности записи, но ограниченный по масштабируемости и вариантам конфигурации мульти-дата-центров.

Требовалось:

- сохранить SLA по p95/p99,
- обеспечить рост пропускной способности,
- формализовать модель данных «таблица-под-запрос»,
- минимизировать простой при миграции.

Исходная инфраструктура Redis включала восемнадцать узлов, каждый из которых имел 4 виртуальных CPU, 60 гигабайт оперативной памяти и 50 гигабайт диска на SATA SSD. Конфигурация кластера строилась на трёх мастер-узлах и трёх репликах для каждого шарда. Для сохранения данных использовалась комбинация RDB и AOF с режимом «каждую секунду», включены четыре потока ввода-вывода пакетная отправка команд в пайп-лайне с глубиной 50. Операционная система — Ubuntu 20.04 LTS с файловой

системой ext4 и планировщиком noop. При пиковой нагрузке утилизация CPU составляла 35–40 %. Такая конфигурация обеспечивала высокую скорость чтения, но из-за однопоточной обработки команд в Redis масштабирование при интенсивных записях было ограничено

Целевая архитектура на Cassandra состояла из восьми узлов, пять из которых находились в первом дата-центре и три во втором. Каждый узел имел 16 vCPU, 32 ГБ оперативной памяти и 500 ГБ NVMe в RAID0. Конфигурация предусматривала коэффициент репликации три в каждом дата-центре, использование уровневой стратегии компакции (LeveledCompactionStrategy) и LZ4-сжатия. Для обеспечения консистентности применялся уровень LOCAL_QUORUM для большинства запросов и LWT с LOCAL_SERIAL для критичных транзакций. Операционная система — Ubuntu 22.04 LTS с файловой системой XFS и планировщиком noop.

III. Требования к архитектуре

- Два дата-центра с независимым обслуживанием нагрузки.
- Настраиваемая консистентность: для обычных операций — LOCAL_QUORUM, для условных операций — LWT (линейризуемые записи) с LOCAL_SERIAL.
- Предсказуемые хвосты задержек под фоновыми работами (compaction, repair).

III.I Топология кластера и хранилище

Кластер Cassandra развёрнут в двух ДЦ с топологией 5+3 узла и фактором репликации RF=3 в каждом ДЦ (NetworkTopologyStrategy). Включены механизмы hinted handoff и read repair. Диски и файловая система: используются NVMe в RAID0 под XFS.

Мотивация выбора: агрегирование пропускной способности на узел при контролируемом MTTR.

Процедура замены предусматривает:

- Безопасное удаление и замена узла кластера (decommission/replace) с контролем потоковой передачи данных;
- Мониторинг по технологии самоконтроля, анализа и отчётности (Self-Monitoring, Analysis and Reporting Technology, SMART);
- Реализация еженедельных snapshot-бэкапы на внешнее хранилище.
- Архитектура хранения JBOD (Just a Bunch Of Disks) рассматривается как альтернатива для ускорения локализации отказа;
- Переход требует оценки среднего времени реагирования (Mean Time to Respond, MTTR) и влияния на оценку производительности и надёжности на уровне p99.

Параметры балансировки кластера Cassandra: количество токенов (num_tokens) указывается явно в конфигурационной таблице);

При загрузке и замене используется намеренное замедления или ограничения скорости передачи данных с целью предотвратить перегрузку системы, сохранить стабильность и эффективно использовать ресурсы

III.II Оптимизация JVM

Используется инструмент HotSpot JVM [7], G1 GC, целевая пауза 10 мс, объём кучи 14 ГБ с включеным COOPs, опора на страничный кэш операционной системы для файловых данных. Медианная пауза сборщика мусора ~4 мс, значение времени p99 ~9 мс на для актуального профиля нагрузки. Сводка общих параметров миграции представлена в Таблице 1.

Параметр	Redis (исходная)	Cassandra (новая)
Версия ПО	Redis 6.2.7 OSS, Cluster mode	Apache Cassandra 4.1.3
Количество узлов	18	8 (5 + 3 в двух ДЦ)
CPU на узел	4 vCPU	16 vCPU
RAM на узел	60 ГБ	32 ГБ
Диск	50 ГБ SATA SSD	500 ГБ NVMe RAID0
Persistence	RDB + AOF everysec	SSTable + LZ4 compression
ОС	Ubuntu 20.04 LTS, ext4, noop	Ubuntu 22.04 LTS, XFS, noop
I/O Threads / Pipeline	4 / batching 50	Многопоточная обработка JVM
Утилизация CPU (пик)	35–40 %	75–85 %

Таблица 1 Сводка параметров миграции

IV. Модель данных и миграция

IV.I Процесс миграции данных

Миграция осуществлялась поэтапно с минимизацией времени простоя системы. Трансформация схемы данных из модели «ключ-значение» [8] Redis в денормализованную модель широких семейств столбцов Cassandra выполнялась с учетом паттернов запросов. Разработка ETL процессов выполнена для преобразования данных с сохранением референциальной целостности. Производилась параллельная работа систем с постепенным переключением трафика чтение/запись на новую конфигурацию системы.

Модель данных была спроектирована с учётом равномерного распределения нагрузки. Ключевое партиционирование осуществлялось по user_id, а сортировка внутри раздела — по временной метке event_ts в порядке убывания. Это обеспечивало быстрый доступ к последним событиям пользователя и выборкам за временной диапазон. Типовые запросы ограничивались выборкой последних N событий и фильтрацией по интервалам времени (Рис.2)

```
CREATE TABLE user_events (
    user_id uuid,
    event_ts timestamp,
    event_type text,
    payload text,
    PRIMARY KEY ((user_id), event_ts)
) WITH CLUSTERING ORDER BY (event_ts
DESC)
```

```
  AND compaction = {'class':
'LeveledCompactionStrategy'}
  AND compression =
{'sstable_compression':
'LZ4Compressor'};
```

Рисунок 2. Типовые запросы

Оптимизация виртуальной машины JAVA включала установку фиксированного размера кучи 14 ГБ с использованием G1GC и целевым временем паузы в 10 мс. Для ускорения работы были включены механизм COOPs и вынесен кеш вне кучи (17 ГБ под страничный кэш операционной системы). В ходе тестирования среднее время паузы сборщика мусора составило 4 мс, значение времени p99 — 9 мс, пропускная способность сборщика мусора — 99,2 % (Таблица 3).

JVM Параметр	Значение
Heap Size	14 ГБ (`-Xms14G -Xmx14G`)
Garbage Collector	G1GC (`-XX:+UseG1GC -XX:MaxGCPauseMillis=10`)
Compressed OOPs	Включены
Off-heap cache	17 ГБ (page cache ОС)
GC Pause (median)	4 мс
GC Pause (p99)	9 мс
GC Throughput	99.2 %

Таблица 3 Параметры оптимизации виртуальной машины

Особое внимание уделялось оптимизации структуры таблиц Cassandra для минимизации количества считывания разделов и обеспечения эффективного использования фильтров Блума. Использование инструментов sstableloader и COPY команд обеспечило эффективную загрузку больших объемов данных с контролем пропускной способности для предотвращения перегрузки кластера.

IV.II Модель «таблица-под-запрос»

Как указано выше, применяется партиционирование по user_id и кластеризация по временному атрибуту в порядке убывания (Рисунок 4).

```
CREATE TABLE user_events (
  user_id       uuid,
  event_ts      timestamp,
  event_type    text,
  payload       blob,
  PRIMARY KEY ((user_id), event_ts)
) WITH CLUSTERING ORDER BY (event_ts DESC)
  AND compaction = {
    'class': 'LeveledCompactionStrategy'
  }
  AND compression = {
    'class': 'LZ4Compressor'
  };
```

Рисунок 4.

IV.III Выбор стратегии компактизации (LCS vs TWCS)

Мы используем стратегию равномерного уплотнения (Leveled Compaction Strategy , LCS), поскольку система выполняет как чтения «последние N» для отдельных пользователей, так и произвольные диапазоны по времени без строгой ретенции/TTL, а также работает под смешанной нагрузкой записи/чтения. В этих условиях LCS снижает расширенное чтение для случайных диапазонов и обеспечивает более стабильные хвосты задержек.

Альтернативная стратегия сокращения временного интервала (Time-Window Compaction Strategy, TWCS) предпочтительна при монотонной записи с явной ретенцией и преобладании запросов «скользящих окон».

При введении строгой ретенции может быть применена стратегия TWCS.

IV.IV *Процесс миграции данных*

Миграция проводилась по схеме: Spark-ETL (трансформация модели «ключ-значение» в широкую таблицу) → генерация сортированных таблиц строк SSTables → массовая загрузка sstableloader с замедлением передачи данных до уровня N МБ/с на узел при M параллельных потоках.

Механизм COPY применялся только для малых порций < 10 ГБ и отладки. Для объёмов производственного масштаба использовался исключительно инструмент sstableloader.

IV.V Верификация *целостности*

Контроль полноты выполнялся в два этапа:

- сопоставление количества строк по ключевым диапазонам;
- выборочная проверка содержимого (случайная выборка ключей, сверка полезной нагрузки).

Внутренние CRC файлов использовались для дополнительного контроля целостности выгрузок.

V. Методика эксперимента

V.I Нагрузочный профиль

В эксперименте использовалась конфигурация нагрузочного профиля:

- Генератор: Yahoo! Cloud Serving Benchmark 0.17.0, профиль A (50 % чтение / 50 % запись) с использованием частотного распределение по закону Ципфа.
- Ключевое пространство: 100 млн ключей;
- Размер записи: 1 КБ.
- Прогрев: 15 мин;
- измерение: 60 мин;
- Параллелизм клиента: 1024 потока;
- Пул соединений — по умолчанию драйвера Cassandra/Redis.

V.II Базовые конфигурации сравниваемых систем

Сравнивалась в ходе эксперимента работа двух альтернативных систем:

Cassandra 4.1.x:

- 2 дата центра (5+3), RF=3 в каждом,
- уровни консистентности:

- чтение/запись — LOCAL_QUORUM;
- условные операции — LWT / LOCAL_SERIAL; ФС — XFS;
- диски — NVMe RAID0;

• Операционная система/сеть — тюнинг TCP и файловых лимитов (ulimit -n ≥ 1 048 576).

Redis 6.2.x:

• режим RDB + AOF everysec,
• I/O-threads включены, pipeline = 50;
• сопоставимые профили операционной системы и файловой системы,
• сетевые параметры симметричны.

V.II Статистическая обработка и доверительные интервалы

Метрики агрегируются по одно-минутным окнам. Доверительные интервалы 95 % оцениваются методом бутстрэп по окнам (10 000 повторных выборок).

Для p95/p99 используются HDR-гистограммы клиента, учитывается количество окон и узлов, участвующих в агрегации.

V.III Анализ производительности и преимуществ архитектуры

Методика нагрузочного тестирования строилась воспроизводимой и основана стандартизированном нагрузочного профиля (п. 2.2.1). Использовались записи размером 1 КБ с общим числом ключей 100 миллионов. Прогрев кластера проводился в течение 15 минут, длительность замера — 60 минут при 1024 параллельных потоках. Использовался драйвер DataStax Java Driver 4.15.0. На уровне операционной системы были заданы параметры vm.swappiness=1, net.core.somaxconn=65535. Лимит открытых файлов определен в миллион дескрипторов (Таблица 5).

Параметр нагрузочного теста	Значение
Генератор нагрузки	YCSB 0.17.0
Workload	YCSB-A (50 % read / 50 % update)
Распределение ключей	Zipfian
Объём данных	100 млн ключей, 1 КБ на запись
Прогрев	15 минут
Продолжительность	60 минут
Кол-во потоков	1024
Драйвер	DataStax Java Driver 4.15.0
Настройки ОС	`vm.swappiness=1`, `net.core.somaxconn=65535`, `ulimit -n 1048576`

Таблица 6. Параметры нагрузочного теста

Нагрузочное тестирование продемонстрировало способность кластера Cassandra обрабатывать целевую нагрузку 100,000 RPS с поддержанием приемлемых значений задержки. Метрики производительности показали:

• 95th percentile latency: менее 8ms для операций чтения
• 99th percentile latency: менее 45ms для операций записи
• CPU utilization: 75-85% под пиковой нагрузкой (против 35-40% в Redis)
• Memory utilization: эффективное использование heap space без проблем сборки мусора
• JVM GC pause time: медианное значение 4ms, 99th percentile 9ms

Распределение нагрузки между дата-центрами обеспечило равномерную утилизацию ресурсов и отсутствие горячих точек. Механизмы read repair и hinted handoff гарантировал итоговую консистентность при временных сбоях узлов.

VI. Сравнительный анализ эффективности ресурсов

Сравнительный анализ показал существенные улучшения ключевых показателей по результатам миграции:

Консолидация инфраструктуры. Снижение количества узлов на 56% (с 18 до 8) при сохранении и улучшении производительности. Общая потребление CPU увеличилось с 72 ядер (18×4) до 128 ядер (8×16), но утилизация возросла с 35% до 80%, обеспечивая реальное увеличение эффективной вычислительной мощности.

Оптимизация памяти. Переход с 1080 ГБ ОЗУ (18×60) на 256 ГБ ОЗУ (8×32) при улучшении производительности благодаря оптимизации виртуальной машины JAVA и более эффективной архитектуре кэширования Cassandra.

Масштабируемость хранения. Увеличение дискового пространства с 900 ГБ (18×50) до 4000 ГБ (8×500) обеспечивает долгосрочный рост данных без необходимости горизонтального масштабирования.

Результаты нагрузочного тестирования показали, что Cassandra при восьми узлах достигала 100 000 запросов в секунду с p99 для записи 45 мс и p95 для чтения 8 мс, тогда как Redis на восемнадцати узлах обеспечивал 60 000 запросов в секунду при p99 для записи 22 мс и p95 для чтения 5 мс. При этом Cassandra работала с более высокой загрузкой CPU (75–85 %), но использовала в 4,2 раза меньше оперативной памяти. Доверительный интервал для RPS составил ±1,2 %, для p99 latency — ±1,8 мс. (таблица 7)

Метрика	Redis (18 узлов)	Cassandra (8 узлов)
p95 read latency	5 мс	8 мс
p99 write latency	22 мс	45 мс
RPS	60 000	100 000
CPU util	35–40 %	75–85 %
RAM total	1080 ГБ	256 ГБ
Disk total	900 ГБ	4 ТБ
Количество узлов	18	8

Таблица 7 Результаты нагрузочного тестирования

Тесты отказоустойчивости показали, что при потере одного узла в первом дата-центре производительность снижалась на 8 % с восстановлением через 30 секунд. При отказе второго дата-центра RPS падал на 12 %, а p99 для записи увеличивался на 40 %. Репликация

между дата-центрами проходила с RTT 8 мс. Восстановление данных выполнялось еженедельно через nodetool repair с автоматизацией в Ansible, бэкапы снимались каждые 6 часов в S3.

Сценарий отказа	Изменение RPS	Изменение p99 записи	Время восстановления
Отказ 1 узла в DC1	-8 %	+0 %	30 с
Отказ DC2	-12 %	+40 %	120 с

Таблица 8. Результаты тестов отказоустойчивости

Миграция данных объёмом 2,1 ТБ была реализована через Spark 3.4.1 с последующей загрузкой SSTable через sstableloader. Скорость загрузки составила 220 МБ/с, ограничением была пропускная способность сети. Переключение write-трафика сопровождалось 14-минутным простоем. Проверка целостности выполнялась по количеству строк и CRC. Для тестов подмножеств данных менее 10 ГБ применялся COPY. (Таблица 9)

Миграционные параметры	Значение
Объём данных	2,1 ТБ
Инструмент загрузки	Spark 3.4.1 + `sstableloader`
Скорость загрузки	220 МБ/с
Время переключения write	14 минут
Проверка данных	row count + CRC

Таблица 9. Параметры тестовой миграции

Сопоставление по ресурсоэффективности показало, что на одно ядро Cassandra обрабатывала на 56 % больше запросов, а на ватт мощности — на 48 % больше. При модельном расчёте капитальных затрат в евро выигрыш составил 22 %. (Таблица 10)

Параметр	Изменение при переходе на Cassandra
RPS на ядро	+56 %
RPS на ватт	+48 %
Экономия CAPEX (евро)	-22 %

Таблица 10. Параметры ресурсоэффективности миграции

V.I Пропускная способность и задержки

Суммарная метрика обработки запросов (Requests Per Second, RPS) для кластера Cassandra достигал ~100 000 запросов/с при времени p95 чтения < 8 мс и времени p99 записи < 45 мс; медианная пауза сборщика мусора ~4 мс, время p99 — ~9 мс.

Для базовой конфигурации Redis наблюдался суммарный RPS порядка ~60 000 запросов/с при меньшем времени p99 записи, но более высокой чувствительности к межузловым отклонениям или задержкам в доставке пакетов данных по сети при росте параллелизма.

VI.II Нормировка эффективности

Выполним нормировку эффективности на ядро. Расчетная формула:

$$RPS\_per\_core = \frac{RPS\_total}{vCPU\_total} \qquad (1)$$

При 60 000 RPS на 72 vCPU для Redis получаем значение ~833 RPS/ядро;

При 100 000 RPS на 128 vCPU для Cassandra — ~781 RPS/ядро.

Результат Cassandra после миграции снижается на 6 % относительно начальной системы Redis. Однако, нам представляется более корректным сравнение при фиксированном уровне утилизации CPU (например, 70 % ± 5 п.п.) и/или по p95/p99 при равной нагрузке.

VI.III Влияние фоновых работ (compaction и repair)

Для оценки устойчивости под фоновыми операциями замеряются p95/p99 в трёх режимах:

(а) «чистое поле»,

(б) активная compaction (ограничение compaction_throughput_mb_per_sec на узел),

(в) repair (incremental, weekly) с ограничением параллелизма и пропускной способности стриминга.

Практически важным критерием является ограничение деградации p99 до ≤ X % и RPS до ≤ Y % относительно «чистого поля».

VI.IV Распределение ширины партиций и tombstones

При распределении по закону Ципфа важны размеры партиций p95/p99 (в количестве строк и МБ), значение метрики SSTables-per-read и доля tombstones на чтение. Эти метрики собираются периодически.

При росте «ширины» партиций рекомендуется ввести дополнительную группировку по времени, например, путем задания параметра bucket = toHour(event_ts) в ключе кластеризации для ограничения количество прочтений на запрос.

VI.V Отказоустойчивость и деградационные сценарии

Были выполнены тесты на отказоустойчивость по различным сценариям:

- Сценарий потери одного узла в дата-центре. При RF=3 и LOCAL_QUORUM операции продолжаются без потери доступности; bootstrap/replace выполняется с ограничением потока данных и мониторингом нагрузки.
- Сценарий потери дата-центра. Кворум в оставшемся дата-центре обеспечивает продолжение обслуживания; после восстановления связи процедура восстановления данных синхронизирует расхождения.

Задержка времени отклика между дата-центрами. Влияние метрики Network Round-Trip Time (RTT) учитывается в выборе локальных уровней консистентности и политике маршрутизации клиента.

VI.VI Ограничения и валидность результатов

Нагрузка YCSB-A (50/50) приближает, но не полностью отражает профиль продуктивных запросов, доля облегченных потоков может отличаться, влияя на p99 записи. Полученные метрики валидны для указанной аппаратной/сетевой конфигурации. Изменения настроек систем Redis или Cassandra могут изменить результаты эксперимента.

Вариация задержки пакетов при передаче данных через сеть и время приема-передачи между дата-центрами в испытаниях были стабильными. В случае их существенного изменения рекомендуется повторная валидация.

Заключение

Результаты подтверждают эффективность предложенного архитектурного решения для высокопроизводительных приложений с требованиями к масштабируемости и отказоустойчивости при одновременной оптимизации использования инфраструктурных ресурсов.

Переход на Apache Cassandra позволил повысить суммарную пропускную способность при сохранении предсказуемых хвостов задержек и обеспечить мульти-ДЦ устойчивость много-датацентровой архитектуры распределённой системы хранения данных.

Миграция позволила сократить количество узлов на 56%, увеличить пиковую пропускную способность на 66%, уменьшить использование оперативной памяти более чем в четыре раза и обеспечить работу в двух дата-центрах без потери SLA. Представленная методика нагрузочного тестирования и параметры конфигурации позволяют воспроизвести результаты на аналогичном оборудовании и могут служить практическим ориентиром для перехода с Redis на Cassandra в системах с высокой долей операций записи.

Библиография

[1] Redis Labs. Redis Documentation. Retrieved from https://redis.io/documentation
[2] Apache Software Foundation. Apache Cassandra Documentation 4.0. Retrieved from https://cassandra.apache.org/doc/latest/
[3] Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), 35-40.
[4] Carpenter, J., & Hewitt, E. (2016). Cassandra: The Definitive Guide, 2nd Edition. O'Reilly Media.
[5] Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional
[6] Redmond, E., & Wilson, J. R. (2012). Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Bookshelf
[7] Oracle Corporation. HotSpot Virtual Machine Garbage Collection Tuning Guide. Retrieved from https://docs.oracle.com/en/java/javase/11/gctuning/
[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 41(6), 205-220.

Статья получена 6 августа 2025.
Гаджилов Гамзат Гаджиевич, выпускник магистратуры ВИШ НИЯУ МИФИ, gadjilov@bk.ru
Хатунов Амгалан Владимирович, выпускник магистратуры ВИШ НИЯУ МИФИ, fanqo@yandex.ru

High-load cluster Migration from Redis to Apache Cassandra: Architectural Solutions and Performance Analysis

G.G. Gadgilov, A.V. Khatunov

***Abstract* – The paper presents a case study of migrating a high-load system from Redis 6.2.x to Apache Cassandra 4.1.x in a configuration of two data centers (RF=3+3, 'LOCAL_QUORUM'). A reproducible load testing methodology (YCSBA, Zipf, 100 million keys, 1 KB write, warm-up/metering) and a comparison of p95/p99 latency and bandwidth are given, taking into account the configurations of the operating system, file structure, disk space and the configuration of the JAVA virtual machine. The results of failure tests for node loss and data center loss are shown. Operating procedures, including recovery and backup, are presented. Options for compromise optimization of the choice of data compaction strategy are discussed with the analysis of alternative approaches, such as the LCS uniform compaction strategy and the TWCS time series strategy. The influence of background compaction tasks and data synchronization between nodes in a high-load distributed data storage system on "latency tails" - temporary jumps in latency leading to system performance degradation under high loads is considered.**

***Keywords*: NoSQL, Apache Cassandra, Redis, data migration, distributed systems, performance, JVM optimization, reproducibility of experiments, fault tolerance**

BIBLIOGRAPHY

[1] Redis Labs. Redis Documentation. Retrieved from https://redis.io/documentation

[2] Apache Software Foundation. Apache Cassandra Documentation 4.0. Retrieved from https://cassandra.apache.org/doc/latest/

[3] Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), 35-40.

[4] Carpenter, J., & Hewitt, E. (2016). Cassandra: The Definitive Guide, 2nd Edition. O'Reilly Media.

[5] Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional

[6] Redmond, E., & Wilson, J. R. (2012). Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Bookshelf

[7] Oracle Corporation. HotSpot Virtual Machine Garbage Collection Tuning Guide. Retrieved from https://docs.oracle.com/en/java/javase/11/gctuning/

[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 41(6), 205-220.