Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Пермский национальный исследовательский**

**политехнический университет»**

Факультет прикладной математики и механики

Кафедра «Вычислительная математика, механика и биомеханика»

Направление подготовки: 09.04.02 «Информационные технологии и системная инженерия»

Отчет

по дисциплине: «Разработка высоконагруженных web-приложений»

Выполнил:

Студент группы ИТСИ-18-1м

Давыдова Ирина Николаевна

Проверил:

Ассистент кафедры ВММБ

Истомин Денис Андреевич

Пермь 2019 г

Оглавление

[Лабораторные работы по дисциплине 3](#_Toc26647183)

[Лабораторная работа №1 3](#_Toc26647184)

[Лабораторная работа №2 5](#_Toc26647185)

[Лабораторная работа №3 7](#_Toc26647186)

[Лабораторная работа №4 10](#_Toc26647187)

[Доклад по дисциплине 15](#_Toc26647188)

[Вывод 24](#_Toc26647189)

[Список литературы 25](#_Toc26647190)

# Лабораторные работы по дисциплине

## Лабораторная работа №1

**Цель:** Получить теоретические знания и практические навыки работы с балансировкой трафика к web-приложениям.

**Задача:** Создать web-приложение на java, с 2 эндпоинтами (инкрементация и чтение счетчика). Развернуть несколько инстанций приложения в Docker. Запустить NGINX в качестве балансировка. Запустить утилиту ab и продемонстрировать работы различных видов балансировки (round robin, ip hash и др.).

**Теория:**

Балансировка нагрузки может быть полезной в случаях, когда ваш один-единственный сервер уже не справляется с возложенными на него задачами в силу возросшего количества запросов. В Nginx роль балансировщика реализует модуль HttpUpstreamModule, обычно по умолчанию содержащийся в сборке Nginx.

Допустим, вы разместили ваше приложение на нескольких серверах, пусть это будут три сервера с именами myserv-1.local, myserv-2.local и myserv-3.local, расположенные в вашей локальной сети (конечно же, на самом деле без разницы, где они будут размещаться). Для того, чтобы описать такое «облако» из трёх серверов в Nginx используется опция upstream:

upstream mySuperCloud {

server myserv-1.local;

server myserv-2.local;

server myserv-3.local;

}

Описанное таким образом «облако» из нескольких upstream-серверов теперь можно использовать в качестве значения параметра [proxy\_pass](http://www.ashep.org/2011/nginx-obratnyj-proksi-server/), рассмотренного в предыдущей статье. Везде, где Nginx будет наталкиваться на ссылку (в нашем случае — «mySuperCloud»), он будет по алгоритму Round-Robin выбирать следующий сервер и выполнять обратное проксирование к нему и от него. Таким образом, теперь, описывая виртуальный хост, вы можете сослаться на ваше «облако»:

server {

listen 192.168.0.1:80;

access\_log /var/log/nginx/proxy.log;

location / {

proxy\_pass http://mySuperCloud;

}

}

Соединения к серверам для балансировки нагрузки могут распределяться по различным правилам. Существуют несколько методов распределения запросов. Я перечислю основные:

* **round-robin** — используется по умолчанию. Веб сервер равномерно распределяет нагрузку на сервера с учетом их весов. Специально указывать этот метод в конфигурации не надо.
* **least-connected** — запрос отправляется к серверу с наименьшим количеством активных подключений. В конфигурации данный параметр распределения запросов устанавливается параметром least\_conn.
* **ip-hash** — используется хэш функция, основанная на клиентском ip адресе, для определения, куда направить следующий запрос. Используется для привязки клиента к одному и тому же серверу. В предыдущих методах один и тот же клиент может попадать на разные серверы.
* **hash** — задаёт метод балансировки, при котором соответствие клиента серверу определяется при помощи хэшированного значения ключа. В качестве ключа может использоваться текст, переменные и их комбинации.
* **random** — балансировка нагрузки, при которой запрос передаётся случайно выбранному серверу, с учётом весов.

В платной версии существуют дополнительный более продвинутый метод распределения нагрузки — **least\_time**, при котором запрос передаётся серверу с наименьшими средним временем ответа и числом активных соединений с учётом весов серверов.

## Лабораторная работа №2

**Цель:** Получить теоретические знания и практические навыки по обеспечению отказоустойчивости web-приложений.

**Задача:** Использовать веб-приложение в docker из предыдущего задания. Запустить Ngninx для организации отказоустойчивости. Запустить утилиту ab и продемонстрировать отказоустойчивость при поочередном отключении экземпляров веб-приложения. Проверить с различными настройками.

**Теория:**

Проблем при балансировки нагрузки с помощью nginx может быть масса. В обычном рабочем проекте нельзя просто взять и разделить нагрузку на несколько серверов. Само приложение, его БД должны быть готовы к этому. Первое, с чем можно столкнуться — как правильно определить, что с сервером проблемы. В бесплатной версии nginx нет никаких инструментов для того, чтобы определить, что ваш бэкенд отвечает правильно.

Допустим, один из серверов перегружен и он отдает неправильные ответы. То есть он жив, отвечает, но ответы нам не подходят. Балансировщик будет считать, что все в порядке, запросы на проблемный сервер будут продолжать идти. Nginx исключит его из списка бэкендов только тогда, когда он полностью перестанет отвечать. Но это лишь малая часть проблем, которые могут приключиться. Чаще всего сервера не отваливаются полностью, а начинают тупить или отдавать, к примеру 502 ошибку. В бесплатной версии nginx будет считать, что все в порядке.

Для того, чтобы анализировать ответ бэкенда и в зависимости от этого ответа, решать, в каком состоянии находится сервер, вам необходим модуль ngx\_http\_upstream\_hc\_module. Он доступен только в коммерческой подписке. С помощью этого модуля можно тестировать код ответа, наличие или отсутствие определённых полей заголовка и их значений, а также содержимое тела ответа. Без этих данных качественно настроить работу балансировщика трудно.

Отмечу еще несколько полезных настроек, на которые надо обратить внимание, при настройке балансировки нагрузки с помощью nginx:

* proxy\_connect\_timeout — задаёт таймаут для установления соединения с проксированным сервером. При отсутствии ответа за указанное время сервер будет считаться неработающим. Дефолтное значение 60 секунд. Это очень много, если у вас большие нагрузки. За минуту скопится огромное количество висящих соединений, которые мог бы обработать другой бэкенд.
* proxy\_read\_timeout — задаёт таймаут при чтении ответа проксированного сервера. Дефолт тоже 60 секунд. Чаще всего имеет смысл уменьшить значение.

Также для обеспечения отказоустойчивости можно указать следующие параметры:

server 10.32.18.6:8080 max\_fails=2 fail\_timeout=10s;

server 10.32.18.7:8080 max\_fails=2 fail\_timeout=10s;

server 10.32.18.8:8080 max\_fails=2 fail\_timeout=10s;

* max\_fails – задаёт число неудачных попыток работы с сервером, которые должны произойти в течение времени, заданного параметром fail\_timeout, чтобы сервер считался недоступным на период времени, также заданный параметром fail\_timeout. Дефолтное значение — 1.
* fail\_timeout - Задаёт время, в течение которого должно произойти заданное число неудачных попыток работы с сервером для того, чтобы сервер считался недоступным и время, в течение которого сервер будет считаться недоступным. По умолчанию параметр равен 10 секундам.

## Лабораторная работа №3

**Цель:** Получить теоретические знания и практические навыки в работе с key-value хранилищами данных.

**Задача:** Использовать веб-приложение в docker из предыдущего задания. Перенести счетчик в redis.

**Теория:**

Redis (расшифровывается как Remote Dictionary Server) – это быстрое хранилище данных типа «ключ‑значение» в памяти с открытым исходным кодом для использования в качестве базы данных, кэша, брокера сообщений или очереди. Проект возник, когда Сальваторе Санфилиппо, первоначальный разработчик Redis, пытался улучшить масштабируемость стартапа в Италии. Redis обеспечивает время отклика на уровне долей миллисекунды и позволяет приложениям, работающим в режиме реального времени, выполнять миллионы запросов в секунду. Такие приложения востребованы в сфере игр, рекламных технологий, финансовых сервисов, здравоохранения и IoT. Redis широко применяется для кэширования, управления сеансами, разработки игр, создания таблиц лидеров, аналитики в режиме реального времени, работы с геопространственными данными, поддержки служб такси, чатов и сервисов обмена сообщениями, потоковой передачи мультимедиа и приложений с отправкой сообщений по модели «издатель – подписчик» (Pub/Sub).

Все данные в Redis хранятся в памяти, а не на дисках или твердотельных накопителях, как в других базах данных. Поскольку Redis, как и другие хранилища данных в памяти, не нуждается в доступе к диску, это исключает задержки, связанные с поиском, и обеспечивает доступ к данным за микросекунды. В число возможностей Redis входит поддержка разнообразных структур данных, обеспечение высокой доступности, работа с геопространственными данными, создание скриптов Lua, проведение транзакций, постоянное хранение данных на диске и поддержка кластеров. Все это упрощает создание приложений, работающих в режиме реального времени в масштабе всего Интернета.

Все данные Redis находятся в основной памяти сервера, в отличие от таких баз данных, как PostgreSQL, Cassandra, MongoDB и других, которые большую часть данных хранят на магнитных дисках или SSD‑накопителях. По сравнению с традиционными дисковыми базами данных, требующими циклического обращения к диску для большинства операций, хранилища данных в памяти, такие как Redis, свободны от этого ограничения. Благодаря этому многократно увеличивается количество выполняемых операций и сокращается время отклика. В результате обеспечивается чрезвычайно высокая производительность. Операции чтения или записи в среднем занимают менее миллисекунды, скорость работы достигает миллионов операций в секунду.

В отличие от упрощенных хранилищ на основе пар «ключ – значение», которые поддерживают ограниченный набор структур данных, Redis поддерживает огромное разнообразие структур данных, позволяющее удовлетворить потребности разнообразных приложений. Типы данных Redis включают:

* строки – текстовые или двоичные данные размером до 512 МБ;
* списки – коллекции строк, упорядоченные в порядке добавления;
* множества – неупорядоченные коллекции строк с возможностью пересечения, объединения и сравнения с другими типами множеств;
* сортированные множества – множества, упорядоченные по значению;
* хэш‑таблицы – структуры данных для хранения списков полей и значений;
* битовые массивы – тип данных, который дает возможность выполнять операции на уровне битов;
* структуры HyperLogLog – вероятностные структуры данных, служащие для оценки количества уникальных элементов в наборе данных.

Redis упрощает код, позволяя писать меньше строк для хранения, использования данных и организации доступа к данным в приложениях. К примеру, если приложение содержит данные, хранящиеся в хэш‑таблице, и требуется сохранить эти данные в хранилище, можно просто использовать структуру данных хэш‑таблицы Redis. Решение подобной задачи с использованием хранилища данных, не поддерживающего структуры хэш‑таблиц, потребует написания серьезного объема кода для преобразования данных из одного формата в другой. Redis уже оснащен встроенными структурами данных и предоставляет множество возможностей их комбинирования и взаимодействия с данными клиента. Разработчикам под Redis доступны более ста клиентов с открытым исходным кодом. Поддерживаемые языки программирования включают Java, Python, PHP, C, C++, C#, JavaScript, Node.js, Ruby, R, Go и многие другие.

В Redis применяется архитектура узлов «ведущий‑подчиненный» и поддерживается асинхронная репликация, при которой данные могут копироваться на несколько подчиненных серверов. Это обеспечивает как улучшенные характеристики чтения (так как запросы могут быть распределены между серверами), так и ускоренное восстановление в случае сбоя основного сервера. Для обеспечения постоянного хранения Redis поддерживает снимки состояния на момент времени (копирование наборов данных Redis на диск).

Redis предлагает архитектуру «ведущий‑подчиненный» с одним ведущим узлом или с кластерной топологией. Это позволяет создавать высокодоступные решения, обеспечивающие стабильную производительность и надежность. Если требуется настроить размер кластера, доступны различные варианты вертикального и горизонтального масштабирования. В результате можно наращивать кластер в соответствии с потребностями.

Redis – проект с открытым исходным кодом, поддерживаемый активным сообществом. Поскольку Redis базируется на открытых стандартах, поддерживает открытые форматы данных и имеет множество клиентов, отсутствует вероятность блокировки поставщиком или технологического тупика.

## Лабораторная работа №4

**Цель:** Получить теоретические знания и практические навыки по партиционированию в СУБД.

**Задача:** Пройти tutorial, продемонстрировать работу партиционирования на СУБД Postrege SQL.

**Теория:**

Партиционировать таблицу — разбить её на несколько физически отдельных частей, которые являются логически связанными. Партиционирование имеет следующие преимущества:

* Скорость выполнения запросов может возрасти в некоторых ситуациях, особенно если большинство запрашиваемых строк находится в одной партиции или в небольшом их числе.
* Когда происходит чтение или обновление большого числа строк внутри одной партиции, скорость запроса может быть увеличена за счёт последовательного сканирования партиции вместо доступа по индексу или случайного доступа к данным по всей таблице.
* Добавление и удаление одновременно большого массива данных может быть выполнено с помощью добавления или удаления партиций, если это было учтено при проектировании. Применение ALTER TABLE DETACH PARTITION или удаление отдельной партиции при помощи DROP TABLE работают быстрее, чем удаление строк по отдельности. Кроме того, эти команды позволяют избежать переполнения VACUUM, вызванного множеством DELETE.
* Редко используемые данные могут быть перенесены в более дешевые и медленные хранилища.

Преимущества технологии будут заметны, только если исходная таблица была очень большой. В каждом конкретном случае выигрыш зависит от самого приложения. Эмпирическое правило гласит, что размер таблицы должен превышать размер физической памяти, выделенной под БД сервера.

В PostgreSQL встроена поддержка следующих видов партиционирования:

* Партиционирование по интервалам. Таблица разбивается на интервалы, задаваемые ключевой колонкой или множеством колонок. Интервалы для каждой партиции не пересекаются. Например, таблица может разбиваться на интервалы по датам или идентификаторам конкретных объектов в зависимости от предметной области и характера хранимых в таблице данных.
* Партиционирование по списку. Таблица разбивается при помощи явного перечисления для каждого ключа, в какой партиции содержится значение.

Если вашему приложению требуются другие формы партиционирования, отличные от представленных выше, могут быть использованы альтернативные методы, такие как наследование и представления с UNION ALL. Эти методы предоставляют вам большую гибкость в настройке, однако они имеют некоторые недостатки по скорости в сравнении со встроенными методами.

PostgreSQL предоставляет пользователям возможность самостоятельно выбрать, каким образом таблица будет разделена на партиции. Таблица, разделенная на партиции, называется партиционированной. Спецификация содержит в себе метод партиционирования и список колонок или выражений, используемых в качестве ключа партиционирования.

Все строки, добавляемые в партиционированную таблицу, будут направлены в одну из партиций в зависимости от ключа партиционирования. Каждая партиция содержит подмножество данных для определённого диапазона ключей. Поддерживаемые в настоящий момент методы партиционирования включают интервалы допустимых значений для ключа и списки, где каждой партиции сопоставлены интервалы ключей или списки ключей соответственно.

В свою очередь, партиции могут быть сами по себе поделены при помощи сабпартиционирования. Партиции могут иметь свои собственные индексы, ограничения и значения по умолчанию, не связанные с другими партициями. Индексы должны создаваться отдельно для каждой партиции. Для дополнительной информации про создание партиционированных таблиц и партиций смотрите раздел CREATE TABLE в официальной документации.

Невозможно превратить обычную таблицу в партиционированную или наоборот. Тем не менее, можно добавить обычную или партиционированную таблицу, содержащую данные, как партицию в партиционированную таблицу. Также можно удалить партицию из патиционированной таблицы, превратив её в обычную таблицу. Для этого обратитесь к подкомандам ATTACH PARTITION и DETACH PARTITION в разделе ALTER TABLE.

Индивидуальные партиции прилинкованы к партиционированной таблице с неявным наследованием. Однако, для партиционированных таблиц и партиций некоторые возможности наследования недоступны. Например, партиция не может иметь других родителей кроме таблицы, от которой она была отделена. Кроме того, обычная таблица не может наследоваться от партиции. Таким образом, партиционированные таблицы и партиции не находятся в отношениях наследования с обычными таблицами. Так как иерархия партиций всё ещё состоит из партиционированных таблиц и партиций, к ним применяются обычные правила наследования, за некоторыми исключениями. Наиболее важные из них:

* Ограничения CHECK и NOT NULL всегда наследуются всеми партициями. Ограничения CHECK, помеченные как NO INHERIT, не могут быть использованы в партиционированных таблицах.
* Можно использовать ONLY для добавления или удаления ограничения в партиционированной таблице, только если она не одержит партиций. Если у таблицы есть партиции, то использование ONLY для добавления или удаления ограничения вызовет ошибку. Вместо этого можно добавлять и удалять ограничения, не представленные в родительской таблице, напрямую в партиции. Так как партиционированные таблицы не содержат данные напрямую, попытки использовать TRUNCATE ONLY на партиционированной таблице будут всегда возвращать ошибку.
* Партиции не могут иметь колонки, не представленные в родительской таблице. Нельзя добавлять колонки при создании партиции через CREATE TABLE, а также после использования ALTER TABLE. Таблицы могут быть добавлены как партиции с помощью ALTER TABLE ... ATTACH PARTITION при условии, что их колонки полностью совпадают с родительскими, включая колонки OID.
* Нельзя удалить ограничение NOT NULL в колонке партиции, если ограничение представлено в родительской таблице.

Партиции также могут быть внешними таблицами, однако на них налагаются некоторые ограничения. Например, данные, добавленные в партиционированную таблицу, не направляются в партиции внешней таблицы.

# Доклад по дисциплине

RAID – это технология, которая позволяет объединить несколько устройств, а именно, жестких дисков, в нашем случае идет что-то вроде их связки. Таким образом, мы повышаем надежность хранения данных и скорость чтения/записи. Возможно и что-то одно из этих функций. Также очень популярен миф, что RAID предназначен для защиты данных, многие настолько верят в это, что забывают про резервное копирование. Но это не так. RAID-массив никоим образом не защищает пользовательские данные, если вы захотите их удалить, зашифровать, отформатировать - наличие или отсутствие RAID вам абсолютно не помешает. Две основных задачи RIAD-массивов - это защита дисковой подсистемы от выхода из строя одного или нескольких дисков и/или улучшение ее параметров по сравнению с одиночным диском.

Весь входящий поток данных разбивается контроллером на блоки определенного размера, которые последовательно записываются на диски массива. Каждый такой блок является минимальной единицей данных, с которой оперирует RAID-контроллер. На схеме ниже мы схематично представили массив из трех дисков (RAID 5).

Каждая шайба на схеме представляет один такой блок, для обозначения которого используют термины: Strip, Stripe Unit, Stripe Size или Chunk, Сhunk Size. В русскоязычной терминологии это может быть блок, "страйп", "чанк". Мы, во избежание путаницы с другой сущностью, предпочитаем использовать для его обозначения термин Chunk (чанк, блок), в тоже время встроенный во многие материнские платы Intel RAID использует термин Stripe Size.

Группа блоков (чанков) расположенная по одинаковым адресам на всех дисках массива обозначается в русскоязычных терминах как лента или полоса. В англоязычной снова используется Stripe, а также "страйп" в переводах, что в ряде случаев способно внести путаницу, поэтому при трактовании термина всегда следует учитывать контекст его употребления.

Каждая полоса содержит либо набор данных, либо данные и их контрольные суммы, которые вычисляются на основе данных каждой такой полосы. Глубиной или шириной полосы (Stripe width/depth) называется объем данных, содержащийся в каждой полосе.

RAID 0 (striping — «чередование») — дисковый массив из двух или более жёстких дисков без резервирования. Информация разбивается на блоки данных Ai фиксированной длины и записывается на оба/несколько дисков поочередно, то есть один блок на первый диск, а второй блок на второй диск соответственно.

Достоинство: скорость считывания файлов увеличивается в n раз, где n — количество дисков. При этом такая оптимальная производительность достигается только для больших запросов, когда фрагменты файла находятся на каждом из дисков.

Недостаток: увеличивается вероятность потери данных: если вероятность отказа 1 диска равна p, то вероятность выхода из строя массива RAID 0 из двух дисков равна 2p-p\*p. Таким образом, если вероятность отказа одного диска за год равна 1 %, то вероятность отказа массива RAID0 из двух дисков составляет 1,99 %, то есть практически в два раза больше. Однако, при отказе одного диска теряется только часть информации.

RAID 1 (mirroring — «зеркалирование») — массив из двух (или более) дисков, являющихся полными копиями друг друга. Не следует путать с массивами RAID 1+0 (RAID 10), RAID 0+1 (RAID 01), в которых используются более сложные механизмы зеркалирования.

Достоинства:

* Обеспечивает приемлемую скорость записи (такую же, как и без дублирования) и выигрыш по скорости чтения при распараллеливании запросов;
* Имеет высокую надёжность — работает до тех пор, пока функционирует хотя бы один диск в массиве. Вероятность выхода из строя сразу двух дисков равна произведению вероятностей отказа каждого диска, то есть значительно ниже вероятности выхода из строя отдельного диска. На практике при выходе из строя одного из дисков следует срочно принимать меры — вновь восстанавливать избыточность. Для этого с любым уровнем RAID (кроме нулевого) рекомендуют использовать диски горячего резерва.

Недостатки:

* Недостаток RAID 1 в том, что по цене двух (и более) жестких дисков пользователь фактически получает объём лишь одного;
* Возможность появления ошибок в данных при использовании двух дисков. При использовании 3 и более ошибки можно устранить.

Массивы такого типа основаны на использовании кода Хэмминга. Диски делятся на две группы: для данных и для кодов коррекции ошибок, причём если данные хранятся на 2n-n-1 дисках, то для хранения кодов коррекции необходимо n дисков. Суммарное количество дисков при этом будет равняться 2n-1. Данные распределяются по дискам, предназначенным для хранения информации, так же, как и в RAID 0, то есть они разбиваются на небольшие блоки по числу дисков. Оставшиеся диски хранят коды коррекции ошибок, по которым в случае выхода какого-либо жёсткого диска из строя возможно восстановление информации. Метод Хэмминга давно применяется в памяти типа ECC и позволяет на лету исправлять однократные и обнаруживать двукратные ошибки.

Достоинством массива RAID 2 является повышение скорости дисковых операций по сравнению с производительностью одного диска.

Недостатком массива RAID 2 является то, что минимальное количество дисков, при котором имеет смысл его использовать — 7, только начиная с этого количества для него требуется меньше дисков, чем для RAID 1 (4 диска с данными, 3 диска с кодами коррекции ошибок), в дальнейшем избыточность уменьшается по экспоненте.

В массиве RAID 3 из n дисков данные разбиваются на куски размером меньше сектора (разбиваются на байты или блоки) и распределяются по n-1 дискам. Ещё один диск используется для хранения блоков чётности. В RAID 2 для этой цели применялся n-1 диск, но большая часть информации на контрольных дисках использовалась для коррекции ошибок «на лету», в то же время большинство пользователей устраивает простое восстановление информации в случае её повреждения, для чего хватает данных, умещающихся на одном выделенном жёстком диске.

Отличия RAID 3 от RAID 2: невозможность коррекции ошибок на лету.

Достоинства:

* Высокая скорость чтения и записи данных;
* Минимальное количество дисков для создания массива равно трём.

Недостатки:

* Массив этого типа хорош только для однозадачной работы с большими файлами, так как время доступа к отдельному сектору, разбитому по дискам, равно максимальному из интервалов доступа к секторам каждого из дисков. Для блоков малого размера время доступа намного больше времени чтения;
* Большая нагрузка на контрольный диск, и, как следствие, его надёжность сильно падает по сравнению с дисками, хранящими данные.

RAID 4 похож на RAID 3, но отличается от него тем, что данные разбиваются на блоки, а не на байты. Таким образом, удалось отчасти «победить» проблему низкой скорости передачи данных небольшого объёма. Запись же производится медленно из-за того, что чётность для блока генерируется при записи и записывается на единственный диск.

Из широко распространённых систем хранения RAID-4 применяется на устройствах компании NetApp (NetApp FAS), где его недостатки успешно устранены за счет работы дисков в специальном режиме групповой записи, определяемом используемой на устройствах внутренней файловой системой WAFL.

Основным недостатком уровней RAID от 2-го до 4-го является невозможность производить параллельные операции записи, так как для хранения информации о чётности используется отдельный контрольный диск. RAID 5 не имеет этого недостатка. Блоки данных и контрольные суммы циклически записываются на все диски массива, нет асимметричности конфигурации дисков. Под контрольными суммами подразумевается результат операции XOR (исключающее или). Xor обладает особенностью, которая даёт возможность заменить любой операнд результатом, и, применив алгоритм xor, получить в результате недостающий операнд. Например: a xor b = c (где a, b, c — три диска рейд-массива), в случае если a откажет, мы можем получить его, поставив на его место c и проведя xor между c и b: c xor b = a. Это применимо вне зависимости от количества операндов: a xor b xor c xor d = e. Если отказывает c, тогда e встаёт на его место и, проведя xor, в результате получаем c: a xor b xor e xor d = c. Этот метод по сути обеспечивает отказоустойчивость 5 версии. Для хранения результата xor требуется всего 1 диск, размер которого равен размеру любого другого диска в RAID.

Минимальное количество используемых дисков равно трём.

Достоинства:

* RAID 5 получил широкое распространение, в первую очередь благодаря своей экономичности. Объём дискового массива RAID 5 рассчитывается по формуле (n-1)\*hddsize, где n — число дисков в массиве, а hddsize — размер диска (наименьшего, если диски имеют разный размер). Например, для массива из четырёх дисков по 80 гигабайт общий объём будет (4 − 1) \* 80 = 240 гигабайт, то есть «потеряется» всего 25 % против 50 % RAID 10. И с увеличением количества дисков в массиве экономия (по сравнению с другими уровнями RAID, обладающими отказоустойчивостью) продолжает увеличиваться;
* RAID 5 обеспечивает высокую скорость чтения — выигрыш достигается за счёт независимых потоков данных с нескольких дисков массива, которые могут обрабатываться параллельно.

Недостатки

* Производительность RAID 5 заметно ниже на операциях типа Random Write (записи в произвольном порядке), при которых производительность падает на 10-25 % от производительности RAID 0 (или RAID 10), так как требует большего количества операций с дисками (каждая операция записи, за исключением так называемых full-stripe write-ов, заменяется на контроллере RAID на четыре — две операции чтения и две операции записи);
* При выходе из строя одного диска надёжность тома сразу снижается до уровня RAID 0 с соответствующим количеством дисков n-1 — то есть в n-1 раз ниже надёжности одного диска — данное состояние называется критическим (degrade или critical). Для возвращения массива к нормальной работе требуется длительный процесс восстановления, связанный с ощутимой потерей производительности и повышенным риском;
* В ходе восстановления (rebuild или reconstruction) контроллер осуществляет длительное интенсивное чтение, которое может спровоцировать выход из строя ещё одного или нескольких дисков массива. Кроме того, в ходе чтения могут выявляться ранее не обнаруженные сбои чтения в массивах cold data (данных, к которым не обращаются при обычной работе массива, архивные и малоактивные данные), препятствующие восстановлению. Если до полного восстановления массива произойдет выход из строя, или возникнет невосстановимая ошибка чтения хотя бы на ещё одном диске, то массив разрушается и данные на нём восстановлению обычными методами не подлежат. Для предотвращения таких ситуаций в RAID-контроллерах может применяться анализ атрибутов S.M.A.R.T.

RAID 6 — похож на RAID 5, но имеет более высокую степень надёжности — три диска данных и два диска контроля чётности. Основан на кодах Рида — Соломона и обеспечивает работоспособность после одновременного выхода из строя любых двух дисков. Обычно использование RAID-6 вызывает примерно 10-15 % падение производительности дисковой группы, относительно RAID 5, что вызвано бо́льшим объёмом работы для контроллера (более сложный алгоритм расчёта контрольных сумм), а также необходимостью читать и перезаписывать больше дисковых блоков при записи каждого блока.

Теперь рассмотрим виды RAID массивов:

* аппаратные (hardware): полностью аппаратный RAID-контроллер проводит все операции самостоятельно (независимо от процессора), и представляет весь RAID-массив в виде одного диска. Такие устройства оснащаются сложными вычислительными процессорами и собственной памятью. Очень часто аппаратные контроллеры выполняются в виде плат расширения для PCIe ×4 или PCIe ×16. Цена такого устройства вполне сопоставима со стоимостью неплохого системного блока;
* программно-аппаратно: часть функций контроллера возлагается на плечи центрального процессора и оперативной памяти посредством драйверов. Как правило, по такому принципу работают все недорогие RAID-контроллеры, включая те, что встраиваются сегодня в материнские платы. Управление данными происходит не на уровне самого «железа», а на уровне микрокода BIOS через драйвер ОС. От сюда и такие понятия как «драйвера» на SATA-RAID (без которых RAID видится как отдельные диски), от сюда и проблемы. Во-первых это несовместимость – сгорела материнская плата, и ищи такую же плату целиком (не контроллер), нужна идентичная плата чтобы добыть свои ценные данные;
* программные(software): набор модулей ядра, вместе с утилитами управления, которые реализуют чисто программный RAID, и не требуют необычной аппаратуры. Подсистема Linux RAID реализована в ядре, как уровень над низкоуровневыми драйверами дисков (для IDE, SCSI и Paraport устройств), и интерфейсом блочных устройств. Файловая система, будь то ext2fs, DOS-FAT, или другая, работает поверх блочного интерфейса. Программный RAID, по своей программной природе, склонен быть более гибким, чем аппаратная реализация. Обратная сторона этого - требуется больше процессорного времени, по сравнению с аппаратной реализацией. Конечно, цена не превзойденная. Кроме того программный RAID имеет одну важную отличительную особенность: он оперирует базируясь на разделах, где несколько отдельных дисковых разделов собираются вместе для создания разделов RAID. В этом отличие от большинства аппаратных решений RAID, которые объединяют вместе целые диски в массив. В аппаратных RAID, факт, что массив RAID - прозрачен для операционной системы, упрощает управление. В программном, гораздо больше конфигурационных опций и вариантов, что запутывает дело.

# Вывод

После прохождения курса «Разработка высоконагруженных web-приложений» были получены теоретические знания и практические навыки по работе с NGINX для обеспечения балансировки трафика и обеспечению отказоустойчивости web-приложений, по работе с Redis для обеспечения быстрого доступа к данным, а также по ускорению работы с СУБД за счет использования партиционирования.

# Список литературы

1. Айвалиотис Д.. Администрирование сервера NGINX. М.: ДМК Пресс. – 2018. – С. 288.
2. Салахалдин Джуба.Основы технологий баз данных: учеб. Пособие. М.: ДМК Пресс, 2019. — 400 с.
3. Пеньчень Хуань, Зуофей Вань. Книга рецептов Redis 4.x. Изд.: Packt Publishing. – 2018. – С. 382.