

< Teach
Me
Skills />

Занятие 30.



Работа с базой данных. Индексы

1

Индекс

- специальная структура данных, создаваемая для ускорения поиска, сортировки и фильтрации информации в базе данных. Представьте, что вы хотите найти конкретную книгу в большой библиотеке. Если книги не упорядочены и нет каталога, вам придется просмотреть все книжные полки, чтобы найти нужную книгу. Но если есть индекс, который содержит информацию о каждой книге и ее местоположении, вы можете быстро найти нужную книгу по индексу
- точно так же работает индекс в базе данных. Когда вы создаете индекс для определенного столбца или набора столбцов, база данных создает отдельную структуру данных, которая содержит отсортированные значения из этих столбцов и ссылки на соответствующие записи в таблице. Благодаря индексу база данных может быстро находить нужные данные, а не просматривать все записи в таблице

Когда нужны индексы

- **Поиск данных:** Индексы позволяют быстро находить данные, ускоряя операции поиска. Если вы хотите найти все записи, где определенный столбец имеет определенное значение, индекс поможет базе данных сразу найти соответствующие записи.
- **Сортировка данных:** Если вы часто сортируете данные по определенному столбцу, индекс поможет выполнить сортировку гораздо быстрее. База данных может использовать уже отсортированные значения из индекса, вместо сортировки данных на лету.
- **Ускорение джоинов:** Когда вы объединяете данные из нескольких таблиц, индекс может помочь ускорить процесс соединения. База данных может использовать индексы для быстрого поиска и объединения соответствующих записей из разных таблиц.

Что важно учесть

- индексы также имеют некоторые недостатки. Они занимают дополнительное место на диске и требуют обновления при вставке, обновлении или удалении данных. Поэтому индексы следует создавать с умом, только для тех столбцов, по которым часто выполняются операции поиска или сортировки.

Как работает индекс

- Индекс в базе данных работает по принципу организации данных в структуру, обеспечивающую более эффективный доступ к информации. Обычно используются два основных типа индексов: B-деревья (B-trees) и хеш-таблицы (hash tables)
- В обоих случаях индекс создается на основе одного или нескольких столбцов в таблице базы данных. При добавлении, изменении или удалении записей в таблице, индекс также обновляется, чтобы отражать актуальное состояние данных. Это может приводить к некоторому дополнительному времени выполнения операций записи, но в целом индексы позволяют значительно ускорить операции поиска и сортировки.

Как работает индекс

В-деревья (B-trees): Это наиболее распространенный тип индекса в базах данных. В-деревья представляют собой сбалансированные деревья, состоящие из узлов, в которых хранятся отсортированные значения ключей и ссылки на соответствующие записи в таблице. Корень дерева содержит ссылки на дочерние узлы, и поиск в индексе начинается с корня и спускается по дереву до тех пор, пока не будет найдено соответствие или пока не будет достигнут конечный листовой узел. В-деревья позволяют эффективно выполнять операции поиска, сортировки и объединения данных.

Как работает индекс

Хеш-таблицы (hash tables): Этот тип индекса использует хеш-функции для преобразования значений ключей в уникальные хеш-коды. Хеш-таблица содержит массив ячеек, в которых хранятся хеш-коды и ссылки на соответствующие записи. При выполнении операций поиска, база данных вычисляет хеш-код и использует его для быстрого доступа к соответствующей ячейке. Хеш-таблицы обеспечивают очень быстрый доступ к данным, но они могут столкнуться с проблемами коллизий, когда два или более значений ключей имеют одинаковый хеш-код.

Как работает индекс

Важно отметить, что не для всех столбцов в таблице необходимо создавать индексы. Решение о том, какие столбцы индексировать, зависит от конкретных запросов и операций, выполняемых с данными. Индексы должны использоваться там, где операции поиска, сортировки или объединения являются частыми и требуют высокой производительности

Как работает индекс

Важно отметить, что не для всех столбцов в таблице необходимо создавать индексы. Решение о том, какие столбцы индексировать, зависит от конкретных запросов и операций, выполняемых с данными. Индексы должны использоваться там, где операции поиска, сортировки или объединения являются частыми и требуют высокой производительности.

Если использовать эту возможность правильно, можно сильно повысить производительность при работе с большими базами данных. А можно и сильно понизить.

Как правило индексы нужно вводить на поля, по которым наиболее часто ведется поиск. Задумываться о создании индекса нужно не ранее, чем когда у вас появится хотя бы 10 тысяч записей. В ином случае заметного результата вы не увидите, ибо преждевременная оптимизация – зло.

Как работает индекс

При вставке новых данных или удалении старых структура сбалансированного дерева будет заново пересчитываться. Собственно, чем больше данных и индексов, тем больше деревьев нужно пересчитать. Представьте ситуацию: у вас есть порядка 20 000 записей и 7 индексов на эту таблицу. То есть, при вставке данных нужно заново пересчитать 7 деревьев, в каждом из которых – по 20 000 записей. Строго говоря, использовать индексы для таблиц, в которые будут часто добавляться/удаляться данные, и вовсе не рекомендуется.

full scan - 6 млн - 1.8с

B-tree - ищет по первой букве, по второй из отфильтрованных и тд. Внутри поиск по буквам работает еще быстрее, потому что он отсортированный и мы ищем бинарным поиском. $\log_2 n$

Предостережения

- Не нужно создавать заранее
- Удалять неиспользуемые индексы
- Не использовать индексы на небольших таблицах
- Исходить из медленных запросов - создавать уникальные индексы под них
- Проверить локально до прода

EXPLAIN ANALYZE - чтобы понять откуда растут ноги, план выполнения запроса, показывает план и реальность

Виды индексов

Кластеризованный:

- Обеспечивает физический порядок по выбранному полю;
- Если у таблицы есть кластеризованный индекс, она называется кластеризованной;
- Нужно не более одного индекса на таблицу;
- В MySQL кластеризованный индекс не задается явно пользователем, так как если вы не определяете PRIMARY KEY для своей таблицы, MySQL находит первый индекс UNIQUE, где все ключевые столбцы – NOT NULL, и InnoDB использует его в качестве кластеризованного индекса.

Виды индексов

Некластеризованный:

- В одной таблице возможно до 999 некластеризованных индексов;
- Содержит указатель на строки с реальными данными в таблице;
- Не обеспечивает физический порядок;
- Для некластеризованных индексов присутствуют отдельные таблицы с отсортированными данными, а именно – одна таблица для одного столбца, на котором индекс, поэтому при запросе данных, не входящих в состав данного поля, будет сначала выполняться запрос к полю в данной таблице, а только затем – дополнительный запрос к строке в изначальной таблице.

```
CREATE INDEX index_name ON table_name(column_name)
```

Виды индексов

Составной индекс – построенный с посыланием на несколько колонок одновременно. Иначе говоря, это комплексный индекс, состоящий из нескольких колонок. Такие индексы используют, когда в одном запросе фигурирует более одной колонки. Создание составного индекса:

```
CREATE INDEX index_name ON table_name(first_column_name,  
second_column_name, third_column_name)
```

Как правило эти индексы используются, когда данные в нескольких столбцах логически взаимосвязаны.

Виды индексов

Покрывающий индекс – это индекс, которого вполне достаточно для ответа на запрос без обращения к самой таблице. Благодаря тому, что не нужно ходить непосредственно в исходную таблицу, а ответить можно, используя только индекс, покрывающие индексы немного быстрее в использовании. При этом не стоит забывать, что чем больше колонок, тем более громоздким и медленным становится сам индекс. Так что злоупотреблять этим не стоит. Выше мы говорили о кластеризованных и некластеризованных индексах, которые могут быть уникальными. Это означает, что никакие две поля не имеют одинаковое значение для ключа индекса. В ином же случае индекс не будет уникальным, ведь несколько строк могут содержать одно и то же значение.

Пример создания уникального некластеризованного индекса:

```
CREATE UNIQUE INDEX index_name ON table_name(column_name)
```

Селективность индекса

Селективность индекса - это мера уникальности значений в столбце или наборе столбцов, на которые создан индекс. Она показывает, насколько эффективно индекс может сужать результаты поиска или фильтрации данных.

Селективность измеряется от 0 до 1 или в процентном соотношении от 0% до 100%. Более высокая селективность означает, что значения в индексируемом столбце или столбцах являются более уникальными, и индекс будет более эффективным при сужении результатов.

Селективность индекса

Рассмотрим пример. Предположим, у нас есть индекс на столбце "город" в таблице с информацией о клиентах. Если в этом столбце всего несколько уникальных значений, например, "Москва", "Санкт-Петербург" и "Екатеринбург", индекс будет иметь высокую селективность. При поиске клиентов, проживающих в определенном городе, индекс позволит базе данных быстро найти нужные записи.

Однако, если столбец "город" содержит множество повторяющихся значений или имеет большой диапазон уникальных значений, например, всякие названия городов по всему миру, селективность индекса будет низкой. В этом случае, использование индекса может быть менее эффективным, поскольку он будет возвращать больше записей при поиске по конкретному городу.

Селективность индекса

Высокая селективность индекса полезна, когда операции поиска или фильтрации основаны на значениях столбцов с высокой селективностью. Это позволяет базе данных сократить количество записей, которые ей нужно просмотреть для получения нужных результатов. В то же время, индексы с низкой селективностью могут быть менее эффективными, и их использование может быть ограничено.

Поэтому при создании индексов важно учитывать селективность индексируемых столбцов и анализировать типичные запросы и операции, выполняемые с данными, чтобы выбирать наиболее эффективные индексы.

Транзакции

2

Транзакция

Транзакция в базе данных представляет собой логическую операцию или последовательность операций, которые должны быть выполнены как единое целое. Транзакция представляет собой единицу работы с данными, которая должна быть выполнена целиком и согласованно, чтобы поддерживать целостность базы данных.

START TRANSACTION;

COMMIT - сохранить

ROLLBACK - откатить

Многие субд используют автоматическую фиксацию транзакций, то есть один запрос - одна транзакция, есть в Postgresql, oracle - выключено по умолчанию

Транзакция

```
BEGIN TRANSACTION

--Инструкция 1
UPDATE Goods SET Price = 70
WHERE ProductId = 1;

--Инструкция 2
UPDATE Goods SET Price = 'Текст'
WHERE ProductId = 2;

COMMIT TRANSACTION
```

100 %

Сообщения

(затронута одна строка)

Сообщение 235, уровень 16, состояние 0, строка 12

Не удалось преобразовать значение типа char к money.

Значение типа char имеет неправильный синтаксис.

Уровни изолированности транзакций

Каждый уровень изолированности разрешает/запрещает определенные действия (возможности): для сохранения целостности данных

READ UNCOMMITTED - транзакции видят р-ты других незавершенных транзакций, изоляции вообще нет

READ COMMITTED - параллельно выполняющиеся транзакции видят только зафиксированные изменения из других транзакций, если транзакция использует данные другой которая закоммитилась в этот момент, значит, мы увидим эти изменения

REPEATABLE READ - не видны изменения UPDATE, DELETE, но видны р-ты insert. Фантомное чтение, в MySQL - уровень по умолчанию (даже не увидим insert), самый хороший.

SERIALIZABLE - самый высокий уровень изоляции, используется редко, сильно ухудшает производительность БД, блокирует чтение

3

ACID

ACID

ACID (Atomicity, Consistency, Isolation, Durability) - это набор свойств, которые обеспечивают надежность и целостность транзакций в базах данных. Каждая из этих характеристик играет важную роль в поддержке правильного выполнения операций базы данных.

Атомарность (Atomicity): Атомарность означает, что транзакция рассматривается как неделимая операция. Это означает, что либо все операции в транзакции должны быть успешно выполнены, либо ни одна из них не должна быть выполнена. Если одна операция в транзакции неудачна, то остальные операции, которые были выполнены ранее, должны быть отменены (откат). Таким образом, база данных остается в целостном состоянии.

ACID

Согласованность (Consistency): Согласованность обеспечивает, что транзакция должна привести базу данных из одного согласованного состояния в другое согласованное состояние. Это означает, что транзакция должна соответствовать определенным правилам и ограничениям базы данных, не нарушая целостность данных. Например, если есть ограничение, что сумма всех счетов в базе данных должна быть равна нулю, то транзакция, которая приводит к нарушению этого ограничения, должна быть отклонена.

ACID

Изолированность (Isolation): Изолированность гарантирует, что параллельные транзакции работают независимо друг от друга. Каждая транзакция должна быть выполнена так, как если бы она выполнялась в изоляции от других транзакций, несмотря на то, что фактически они могут выполняться параллельно. Это предотвращает нежелательные эффекты, такие как потеря данных или несогласованное чтение данных из других транзакций.

Прочность (Durability): Прочность гарантирует, что результаты выполненных транзакций остаются постоянными и не теряются после завершения транзакции или сбоя системы. Зафиксированные изменения должны быть сохранены в постоянном хранилище, чтобы даже при возникновении сбоев данные оставались доступными и неповрежденными.

ACID

Свойства ACID крайне важны для поддержания целостности и надежности данных в базах данных, особенно в критических ситуациях, где отказ базы данных может привести к потере данных или несогласованным состояниям. ACID гарантирует, что транзакции выполняются надежно и согласованно, что делает базы данных надежными и предсказуемыми.

Однако, следует отметить, что соблюдение всех свойств ACID может привести к некоторому снижению производительности системы. Поэтому в некоторых случаях, когда требуется более высокая производительность за счет некоторой потери ACID-свойств, могут быть использованы различные уровни изоляции транзакций.

4

Django + PostgreSQL

Как подключить

ElephantSQL

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'your_db_name',
        'USER': 'postgres',
        'PASSWORD': 'your_db_password',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

```
pip install psycopg2
```

```
python manage.py migrate
```

5

Шардирование и партицирование

Шардирование

- **Шардирование (Sharding):** Шардирование представляет собой процесс разделения данных на несколько физических узлов (шардов), где каждый шард содержит подмножество данных. Каждый шард может находиться на отдельном сервере или устройстве хранения данных. Это позволяет распределить нагрузку и обработку данных между несколькими узлами, что увеличивает производительность и способность к обработке больших объемов данных.
- Пример: Представьте, что у вас есть база данных с информацией о клиентах по всему миру. Чтобы улучшить производительность, вы можете разделить данные на шарды в зависимости от географического расположения клиентов. Каждый шард будет содержать данные только для клиентов из определенного региона. Таким образом, при выполнении запросов, связанных с конкретным регионом, база данных будет обращаться только к соответствующему шарду, что улучшит производительность и снизит нагрузку на отдельные узлы.

Партицирование

- **Партицирование (Partitioning):** Партицирование также относится к разделению данных, но внутри одной базы данных или таблицы. При партицировании данные разделяются на логические группы (партиции) на основе определенного критерия, например, диапазона значений, хеш-функции или списков значений. Каждая партиция может храниться на одном узле или диске, что позволяет эффективно управлять большими объемами данных.
- Пример: Предположим, у вас есть таблица с заказами, и вы решаете партицировать ее по дате заказа. Вы можете создать партиции, например, для каждого месяца или года, где данные для каждого периода будут храниться в отдельной партиции. Это упростит выполнение запросов, связанных с конкретными периодами времени, таким образом, запросы будут ограничены только на соответствующие партиции, что повысит производительность.

Пример

```
CREATE TABLE Orders (
    order_id INT,
    order_date DATE,
    customer_id INT,
    order_total DECIMAL(10,2)
) PARTITION BY RANGE (order_date) (
    PARTITION p1 VALUES LESS THAN ('2022-01-01'),
    PARTITION p2 VALUES LESS THAN ('2023-01-01'),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

6

Сигналы в django

Signals

- **Сигналы (signals) в Django** представляют собой механизм, который позволяет отправлять и принимать оповещения (сигналы) при наступлении определенных событий в вашем приложении. Они позволяют связывать различные части приложения, даже если они не имеют явной зависимости друг от друга. Сигналы позволяют выполнить действия в ответ на определенные события без явного вызова из кода.
- В Django сигналы реализованы с помощью системы сигналов Django, которая предоставляет несколько предопределенных сигналов, а также возможность создания собственных сигналов. Некоторые типичные события, которые могут порождать сигналы в Django, включают создание, обновление или удаление объектов модели, аутентификацию пользователей, изменение состояния сеанса и т.д.

Привязка функции к сигналу

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from myapp.models import MyModel

@receiver(post_save, sender=MyModel)
def my_model_post_save(sender, instance, created, **kwargs):
    # Выполнение действий после сохранения объекта MyModel
    if created:
        # Действия, выполняемые при создании объекта
        pass
    else:
        # Действия, выполняемые при обновлении объекта
        pass
```

Отправка сигнала

```
from django.dispatch import Signal

my_signal = Signal()

def my_function(sender, **kwargs):
    # Обработка сигнала
    pass

# Где-то в коде, когда необходимо отправить сигнал
my_signal.connect(my_function)
my_signal.send(sender=None)
```

Пример

- В этом примере создается собственный сигнал `my_signal` с помощью класса `Signal`. Затем функция `my_function` привязывается к сигналу с помощью метода `connect`. При вызове `my_signal.send()`, сигнал отправляется, и все связанные функции выполняются.
- С помощью сигналов Django вы можете реализовывать различные функции, такие как очистка кэша, отправка уведомлений, выполнение дополнительных действий после сохранения объектов модели и других событий. Сигналы предоставляют гибкий и расширяемый способ связи между различными компонентами вашего приложения



7

manage commands

management/commands/ mycommand.py

```
from django.core.management.base import BaseCommand

class Command(BaseCommand):
    help = 'Описание вашей команды'

    def handle(self, *args, **options):
        # Логика вашей команды
        print('Команда успешно выполнена!')
```

```
from .commands import mycommand
```

management/__init__.py

8

Что если приложение
очень долго работает

НА ПОДУМАТЬ



9

django vs flask vs fastapi

Django

Плюсы:

- Полноценный фреймворк, предоставляющий множество инструментов и функциональность для разработки веб-приложений любого масштаба.
- Встроенная аутентификация, авторизация и управление пользователями.
- ORM (Object-Relational Mapping) для работы с базами данных.
- Административный интерфейс, который может быть быстро настроен для управления данными в приложении.
- Широкая и активная сообщество, множество сторонних пакетов и расширений.

Минусы:

- Тяжеловесный по сравнению с другими фреймворками, из-за своего всестороннего подхода и большого набора функций.
- Менее гибкий по сравнению с более легковесными фреймворками, так как часто требует следования конвенциям и предопределенным структурам.
- Возможно, избыточность некоторых функций для небольших и простых проектов.

Flask

- **Плюсы:**

- Легковесный и минималистичный фреймворк, который предоставляет основные инструменты для разработки веб-приложений.
- Гибкость и свобода в выборе архитектуры и компонентов приложения.
- Отлично подходит для небольших и простых проектов, где требуется меньше функциональности.
- Большое сообщество и множество расширений и пакетов, позволяющих добавлять нужный функционал по мере необходимости.

- **Минусы:**

- Отсутствие некоторых функций, которые предлагает Django, требующих дополнительного кода или сторонних пакетов.
- Отсутствие встроенной ORM (хотя можно использовать сторонние ORM).
- Возможно, большая гибкость и свобода может привести к разрозненности и неоднородности в структуре проекта.

FastAPI

- **Плюсы:**

- Быстрый и эффективный фреймворк для разработки веб-приложений с использованием преимуществ современного Python и асинхронности.
- Очень высокая производительность благодаря использованию `async/await` и современных технологий.
- Автоматически генерирует документацию API с помощью стандарта OpenAPI (Swagger).
- Поддержка типизации и автоматической валидации данных.
- Легкая интеграция с существующими системами и фреймворками.

- **Минусы:**

- Возможно, избыточность для простых проектов, где не требуется высокая производительность или асинхронность.
- Относительная новизна фреймворка, что может привести к меньшему количеству сторонних пакетов и документации по сравнению с Django или Flask.
- Может быть сложностями в освоении для новичков в асинхронном прог

Спасибо за внимание!