МИСПИСИТ 4 курс

1. Организация хранения данных в современных СУБД, секционирование таблиц.

- Журналы транзакций
- Файлы данных
- Страница
- Экстент
- Таблицы БД
- Секционирование
  - по диапазону
  - по списку
  - по хэшу
  - по составному ключу
- Как создать секционированную таблицу в SQL?

---

2. Принципы индексации данных, рекомендации по выбору индексов, операторы языка SQL для создания и удаления индексов.

- Принципы индексации данных
- Структуры индексов
  - B-дерево
  - Хэш-индексы
  - Битовые индексы
- Организация индекса
- Составные индексы
- Рекомендации по выбору индексов
- Операторы языка SQL для создания индексов
  - CREATE INDEX
  - CREATE UNIQUE INDEX
  - CREATE CLUSTERED INDEX
- Операторы языка SQL для удаления индексов
  - DROP INDEX

---

3. Организация и использование кластерных индексов.

- Что такое кластерный индекс?
- Отличие от некластерного
- Структура кластерного индекса
- Физическое хранение
- Типы запросов
- Выбор столбцов
- Влияние на производительность
- Вставка данных
- Удаление данных
- Пример в SQL Server
- Рекомендации

---

4. Организация и использование некластерных индексов.

- Что такое некластерный индекс?
- Отличие от кластерного
- Структура некластерного индекса
- Физическое хранение
- Типы запросов
- Выбор столбцов
- Влияние на производительность
- Рекомендации
- Примеры
    - Создание некластерного индекса
    - Включенные и покрывающие индексы

---

5. Специальные виды индексов (filtered, columnstore, hash, memory-optimized).

- Некластерный индекс с INCLUDE
- Фильтруемый индекс (FILTERED)
- Колоночный индекс (COLUMNSTORE)
- Хэш-индекс (HASH)
- Оптимизированный для памяти индекс (MEMORY-OPTIMIZED)

---

6. Планы выполнения запросов

- Query Parsing
- Query Optimization
- Query Execution
- Выборка результатов
- Сортировка и группировка, выполнение агрегаций
- Алгоритмы обработки данных
  - NESTED LOOPS
  - HASH MATCH
  - MERGE
- Просмотр плана выполнения запроса
- Предполагаемый план выполнения
- Действительный план выполнения
- Статистика активных запросов
- Hints в MS SQL
  - Использование индекса
  - Сплошное сканирование индексов
  - Использование алгоритма

---

7. Статистика для оптимизации запросов

- Инструменты и команды для работы с статистикой
- Оптимизатор запросов
- Стоимость выполнения
- Кратность
- Таблица systabsstats
- Таблица sysstatistics
- Пример использования статистики
- Просмотр статистики в MS SQL
- AUTO_CREATE_STATISTICS
- AUTO_UPDATE_STATISTICS
- Именованная статистика
- Отфильтрованная статистика
- Зачем нужна статистика?

---

8. Транзакции в базах данных, операторы языка SQL для управления транзакциями.

- BEGIN TRANSACTION

- COMMIT
- ROLLBACK
- SAVE
- RELEASE SAVEPOINT
- SET TRANSACTION
- Роль транзакций
- Свойства транзакции (ACID)
- Режимы транзакций
  - Неявные транзакции
  - Явные транзакции
- Рекомендации по использованию транзакций
- Вложенные транзакции
- Состояния транзакций
- Уровни изоляции транзакций

---

9. Проблемы многопользовательского доступа к данным, их решение с помощью блокировок.

- Основные проблемы многопользовательского доступа
  - "Грязное" чтение
  - Неповторяющееся чтение
  - Фантомное чтение
- Блокировки как решение
  - Типы блокировок
    - Блокировка на чтение
    - Блокировка на запись
    - Обновляющая блокировка
  - Уровни изоляции транзакций
    - READ UNCOMMITTED
    - READ COMMITTED
    - REPEATABLE READ
    - SERIALIZABLE
  - Стратегии блокировки
    - Пессимистичные блокировки
    - Оптимистичные блокировки
- Пример управления блокировками
- Решение проблем
  - X-Lock

    - S-Lock
    - Алгоритм двухфазного блокирования
- Проблема тупиков
- Влияние разработчика на механизм обработки тупиков
    - Приоритет сеанса
    - Периодичность проверки возникновения тупика
- Практическое использование блокировок

---

10. Уровни изоляции транзакций

- Основные уровни изоляции
    - READ UNCOMMITTED
    - READ COMMITTED
    - REPEATABLE READ
    - SERIALIZABLE
- Пример применения уровня изоляции
- Выбор уровня изоляции
- Рекомендации по выбору уровня изоляции
- Установка уровня изоляции транзакций
- Задание уровня изоляции для отдельного запроса или таблицы

---

11. Уровни схемы и виды блокировок

- Уровень 1 (таблицы)
- Уровень 3 (строки)
- Схемы блокировки (Sybase)
    - Блокировка всех страниц
    - Блокировка страниц данных
    - Блокировка строк данных
- Виды блокировок
    - Совмещаемая блокировка (Shared)
    - Монопольная блокировка (Exclusive)
    - Блокировка обновления (Update)
    - Блокировки с намерением (Intent)
    - Блокировки схем (Schema)
        - Блокировка изменения схемы (Sch-M)
        - Блокировка стабильности схемы (Sch-S)

        - Блокировка массового обновления (Bulk Update)
        - Блокировка диапазона ключа (Key Range)
    - Примеры использования блокировок
    - Эскалация блокировок

---

12. Оптимизация запросов

- Этапы оптимизации запросов
- Приемы оптимизации на этапе предварительной обработки
    - Преобразование инструкций в эквивалентные
    - Преобразование выражений
    - Транзитивное замыкание аргументов поиска
    - Транзитивное замыкание соединений по равенству
    - Преобразование предикатов запросов
- Рекомендации по выполнению запросов
    - Избегать использования функций и вычислений в условиях запросов
    - Избегать соединений по столбцам, требующих преобразования типов
    - Использовать как можно больше аргументов поиска
    - Заменять условия `>` на `>=`
    - Предпочтение запросам, реализуемым одним оператором
- Прямые и отложенные обновления
    - Прямые обновления
        - Обновления на месте
        - Дешевые прямые обновления
        - Дорогие прямые обновления
    - Отложенные обновления
- Рекомендации для ключей индекса

---

13. Определение стоимости плана выполнения запроса

- Что такое план выполнения?
- Определение стоимости плана выполнения запроса
- Оценка стоимости выполнения запроса
- Стоимость сканирования таблицы с блокировкой всех страниц
- Оценка стоимости использования индекса
- Стоимость выполнения точечного запроса с использованием индекса
- Стоимость запроса по диапазону с использованием кластерного индекса

- Соединение таблиц
  - Вложенные циклы (Nested Loop)
  - Слияние (Merge Join)
  - Хэширование (Hash Join)
- Рекомендации по выполнению запросов

---

14. Абстрактные планы запросов

- Инструменты и использование
- Интерпретация абстрактных планов
  - Сканирование
    - Table Scan
    - Index Scan
  - Соединения
    - Nested Loop Join
    - Hash Join
    - Merge Join
  - Агрегация и сортировка
  - Фильтрация
- Язык АПЗ
  - Операторы
    - join
    - union
    - scan
    - store
    - nested
  - Директивы
    - sequence
    - hints
    - table
    - work_t
- Типовые случаи использования АПЗ
  - Указание способа доступа
  - Указание порядка и способа соединения
  - Указание на использование индексов без задания последовательности соединения таблиц
  - Указание на использование рабочих таблиц
- Другие способы использования АПЗ

- Создание плана директивой CREATE PLAN
- Включение/выключение автосохранения планов выполнения запросов
- Включение/выключение связывания запросов с хранимыми АПЗ

1. Организация хранения данных в современных СУБД, секционирование таблиц

Современные системы управления базами данных (СУБД) предоставляют различные механизмы для организации хранения данных, которые влияют на производительность запросов, обслуживание и масштабируемость. Одним из ключевых инструментов для эффективной организации данных является секционирование таблиц (partitioning).

Используемое СУБД для работы с БД дисковое пространство делится на области для журналов транзакций и хранения данных. **Журналы транзакций** необходимы для восстановления БД и записываются последовательно, без чтения в обычном режиме. Для интенсивно используемых БД рекомендуется размещать журналы и данные на разных физических дисках, чтобы избежать затрат времени на перемещение головок диска.

Файлы данных организуются в группы, с первичной файловой группой PRIMARY и возможными дополнительными группами. Логически БД состоит из схем, таких как dbo, sys, information_schema, которые группируют объекты БД.

Для хранения данных используется страничная организация контейнеров, где **страница** имеет фиксированный размер 8 Кб, из которых 8060 байт отводится для данных. Страницы различаются по назначению и включают заголовок, строки данных и таблицу смещения строк. Смежные страницы объединяются в **экстенты**, которые являются основными единицами организации пространства. Экстенты бывают однородными и смешанными.

Таблицы БД по умолчанию имеют единственную секцию, но могут быть секционированными, где группы строк разнесены по секциям на основе функции секционирования. Секции могут храниться в разных файловых группах, но таблица рассматривается как логически единая сущность.

Страницы могут различаться по назначению: для хранения данных, индексов, больших объектов (текста, изображений, данных в формате XML). На странице файла с данными сначала размещается заголовок, за ним расположены строки данных, в конце страницы находится таблица смещения строк. Таблица смещения содержит одну запись для каждой строки на странице, указывая, насколько далеко от начала страницы находится первый байт строки. Записи в таблице смещения строк находятся в обратном порядке относительно последовательности строк на странице.

Экстенты являются основными единицами организации пространства для хранения данных. В SQL Server экстент объединяет восемь страниц, физически размещенных подряд. В связи с тем, что в БД могут встречаться небольшие объекты, такие как

маленькая таблица, SQL Server имеет два типа экстентов: однородные и смешанные экстенты.

Секционирование — это процесс разбиения таблицы на более мелкие, управляемые части, называемые секциями (partitions), которые логически остаются частью той же таблицы, но физически хранятся отдельно.

1. Типы секционирования:
    - Секционирование по диапазону (Range Partitioning): Данные распределяются по определенному диапазону значений. Например, по датам.
    - Секционирование по списку (List Partitioning): Данные делятся на основе предопределенного списка значений.
    - Секционирование по хэшу (Hash Partitioning): Данные распределяются на основе хэш-функции, использующейся для равномерного распределения строк.
    - Секционирование по составному ключу (Composite Partitioning): Комбинируется несколько методов секционирования, например, сначала по диапазону, затем по хэшу.

```
CREATE TABLE sales (
  sale_id INT,
  sale_date DATE,
  amount DECIMAL(10,2)
)
PARTITION BY RANGE (YEAR(sale_date)) (
  PARTITION p0 VALUES LESS THAN (2010),
  PARTITION p1 VALUES LESS THAN (2015),
  PARTITION p2 VALUES LESS THAN (2020)
);
```

2. Принципы индексации данных, рекомендации по выбору индексов, операторы языка SQL для создания и удаления индексов.

Индексация данных в базах данных — это метод оптимизации, который позволяет ускорить выполнение запросов за счет уменьшения количества операций чтения, необходимых для нахождения данных. Индекс в базах данных обычно организуется с использованием структур данных, таких как B-деревья или хэш-таблицы.

Принципы индексации данных основаны на использовании двоичного поиска в отсортированных массивах, что значительно быстрее линейного поиска. Двоичный поиск сокращает количество операций сравнения до log2(N), тогда как линейный поиск требует N/2 операций.

1. **Структуры индексов:**
   - **B-дерево (B-tree):** Наиболее распространенный тип индексов, поддерживает упорядоченные операции, такие как диапазонные запросы.
   - **Хэш-индексы (Hash Indexes):** Оптимальны для точного поиска, но не поддерживают диапазонные запросы.
   - **Битовые индексы (Bitmap Indexes):** Тип индексов, использующийся в колоночных базах данных для атрибутов с низкой кардинальностью.
2. **Организация индекса:** Индекс может быть кластерным, что означает, что он определяет физический порядок данных в таблице. В некластерном индексе хранится ссылка на физическое местоположение строк.
3. **Составные индексы:** Включают несколько колонок и используют их совместно для индексации данных, полезны для сложных запросов.

Индексы ускоряют выборку данных, но требуют дополнительной памяти и времени при модификации данных. Основной эффект от использования индексов — минимизация количества страниц, читаемых из внешней памяти.

Рекомендации

- Создавать индексы для полей, используемых в условиях отбора (where), сортировке (order by), отборе диапазонов (between … and), и соединении таблиц (join ... on).
- Не создавать индексы для полей, по которым не происходит отбора и сортировки, для полей с малым количеством различных значений (пол, группа крови), и для таблиц с малым количеством записей.

- Индексы не создаются для столбцов типов bit, text, image, так как эти типы данных не поддерживают индексацию.

Операторы языка SQL для создания и удаления индексов

1. **Создание индекса:**
   - CREATE INDEX: Стандартный способ создания некластерного индекса.
   - CREATE UNIQUE INDEX: Создает индекс, который запрещает дублирование значений в колонках индексов.
   - CREATE INDEX idx_name ON table_name (column1, column2): Создает индекс idx_name на указанных колонках таблицы table_name.

```
CREATE INDEX idx_customer_name ON customers (last_name, first_name);
```

2. **Создание кластерного индекса (в некоторых СУБД, таких как SQL Server):**

```
CREATE CLUSTERED INDEX idx_order_date ON orders (order_date);
```

3. **Удаление индекса:**
   - DROP INDEX: Используется для удаления индекса.
   - В MySQL: DROP INDEX idx_name ON table_name:

```
DROP INDEX idx_customer_name ON customers;
```

- **Создание индекса:**

```
CREATE [ unique ] [ clustered | nonclustered ] INDEX <имя индекса>
ON <имя таблицы> ( <имя столбца 1> [ asc | desc ] [ , … ] )
[ WITH { fillfactor | max_rows_per_page } = n ]
GO
```

  - **Уникальные индексы** не допускают дублирования значений.
  - **Кластерные индексы** физически упорядочивают записи таблицы, позволяя только один кластерный индекс на таблицу.
  - **Некластерные индексы** могут быть созданы в количестве до 249.
  - **Составные индексы** включают несколько столбцов таблицы.
  - Параметры **fillfactor** и **max_rows_per_page** задают процент заполнения страниц индекса и количество строк на странице соответственно.
- **Удаление индекса:**

```
DROP INDEX <имя таблицы>.<имя индекса>
GO
```

- **Перестройка индекса** для устранения фрагментации:

```
ALTER INDEX <имя индекса> ON <имя таблицы> REBUILD
GO
```

- **Информация о существующих индексах:**

```
sp_helpindex <имя таблицы>
GO
```

Сканирование индекса более эффективно, чем сканирование таблицы, так как строки индекса короче, и считывание одной страницы индекса предоставляет информацию о большем количестве строк.

Индексация данных в СУБД — это балансирование между улучшением производительности запросов и издержками в виде памяти и времени на обновление данных.

3. Организация и использование кластерных индексов.

Кластерные индексы — это особый тип индексов, которые определяют физический порядок строк в таблице. Тип индекса в СУБД с древовидной структурой, где значения индекса вместе с данными хранятся в виде упорядоченного дерева, обычно сбалансированного дерева поиска.

Структура кластерного индекса

Структура: Кластерный индекс организует данные в B-дерево (обычно это B+-дерево) по ключевым значениям. Листья B-дерева содержат физические строки данных, что означает, что строки упорядочены в соответствии с кластерным ключом.

- **Корень (root):** Вершина дерева, содержит отсортированные значения индекса и указатели на дочерние уровни, предоставляя начальную точку поиска.
- **Промежуточные уровни (intermediate):** Хранят отсортированные значения индекса и указатели на дочерние уровни (листья).
- **Листья (leaf):** Самые нижние уровни дерева, содержащие полные строки таблицы (колонки с данными), хранятся в упорядоченном по индексу, фрагментированном виде на диске.

Физическое хранение:

- Поскольку кластерный индекс определяет физический порядок данных в таблице, возможно существование только одного кластерного индекса на таблицу.
- Данные таблицы фактически «встроены» в индекс. Это отличается от некластерных индексов, где индекс содержит лишь указатели на данные.

Использование

1. **Типы запросов:**
   - Эффективны для диапазонных запросов, таких как BETWEEN, благодаря порядку данных.
   - Запросы, которые возвращают большие значительно определенные объемы данных, извлекают выгоду, так как чтение выполняется последовательно.
2. **Выбор столбцов:**
   - Столбцы с высокой уникальностью (высокой кардинальностью), такие как первичный ключ, часто являются хорошими кандидатами для кластерных индексов.
   - Столбцы, которые часто используются в фильтрах запросов или сортировке.
3. **Влияние на производительность:**

- Улучшение производительности чтения как для запросов по отдельным записям, так и по диапазонам данных.
- Ухудшение производительности вставок, обновлений и удалений, так как данные требуют переформатирования для поддержания порядка.

Вставка данных:

1. Если на странице есть место, новая строка помещается на нужную позицию, а записи, большие ее, сдвигаются вниз.
2. Если на странице нет места, выделяется новая страница, и на нее переносится одна половина записей. **Расщепление страниц** является затратной операцией, требующей распределения памяти, переноса строк, корректировки указанных показателей и перестройки некластерных индексов. При интенсивной вставке рекомендуется устанавливать **fillfactor** <> 100%.

Удаление данных:
При удалении последней записи на странице, она переносится в список свободных страниц.

Пример в SQL Server:

В SQL Server кластерный индекс создается следующим образом, часто при создании первичного ключа:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY CLUSTERED,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    HireDate DATE
);
```

Здесь EmployeeID автоматически становится кластерным индексом, так как он определяет физический порядок хранения строк в таблице.

Рекомендации

- Выбирайте кластерный индекс на основании анализа запросов, чтобы максимизировать преимущества от вписывания в порядок хранения.
- Необходимо учитывать, что изменения в таблице могут быть медленнее из-за затрат на переиндексацию.
- Следует избегать частых изменений в кластерной колонке для минимизации накладных расходов при обновлении.
- Создавать для столбцов, являющихся первичными ключами.
- Создавать для столбцов, по которым осуществляется сортировка.

- Создавать для столбцов, по которым осуществляется отбор диапазонов.
- Создавать для столбцов, по которым осуществляется соединение таблиц.
- Создавать для столбцов, которые нечасто изменяются.

4. Организация и использование некластерных индексов.

Некластерный индекс — это тип индекса в СУБД, который создает дополнительную структуру данных для таблицы по одной или нескольким колонкам для ускоренного поиска. В таком индексе хранятся данные индексируемой колонки и **кластерный ключ** индекса. Некластерный индекс не влияет на порядок физического расположения записей, а является вспомогательным инструментом для ускорения поиска данных.

Организация

1. **Структура:**
    - Некластерный индекс организуется в виде B-дерева. Каждый узел дерева содержит ключи индекса и указатели на соответствующие строки в таблице.
    - В отличие от кластерного индекса, не требуется изменять внутреннюю структуру таблицы; данные остаются в исходном, неупорядоченном виде.
2. **Физическое хранение:**
    - Листовые узлы содержат копии ключевых значений и указатели на строки данных. Некластерные индексы могут быть размещены отдельно от данных, что позволяет создавать несколько таких индексов для одной таблицы.

Структура некластерного индекса:*

- На листьях древовидной структуры хранятся указатели на записи кластерного индекса. Указатели необходимы для получения данных других колонок, не находящихся в некластерном индексе. В этом случае СУБД выполнит дополнительный поиск строки по кластерному индексу и достанет значения этих колонок из листьев кластерного индекса, что приводит к двум поискам по B-Tree дереву: первый некластерный и второй кластерный.

Использование некластерных индексов

1. **Типы запросов:**
    - Некластерные индексы эффективны для запросов, которые фильтруют или выбирают данные не по кластерному ключу. Они ускоряют доступ ко второстепенным данным.
    - Полезны для разнообразных запросов, таких как запросы с WHERE, JOIN, и даже ORDER BY или GROUP BY.
2. **Выбор столбцов:**
    - Часто проиндексированные столбцы: те, которые часто используются в

поисках, фильтрах и соединениях.
    - Столбцы с высокой кардинальностью или те, для которых актуальна уникальность и селективность.
3. **Влияние на производительность:**
    - Повышение скорости выборки данных за счет уменьшения количества дисковых операций.
    - Вставки, обновления и удаления занимают больше времени из-за необходимости поддержания дополнительной структуры индексов.

Рекомендации

Рекомендации по использованию некластерных индексов:

- Создавать некластерные индексы для столбцов, не включенных в кластерный индекс.
- Стремиться к тому, чтобы индекс **покрывал запрос** — содержал все используемые в запросе ключи поиска.
- Осуществлять **префиксное сканирование индекса** (обход по дереву).

Покрывающий индекс — это индекс, который содержит все столбцы, необходимые для выполнения запроса, что позволяет избежать дополнительного поиска по кластерному индексу.

Префиксное сканирование индекса — это метод обхода дерева индекса, который позволяет эффективно использовать индекс для выполнения запросов, особенно когда запрос включает префиксные части ключей индекса.

Примеры

Создание некластерного индекса:

```
CREATE INDEX idx_employee_lastname ON Employees (LastName);
```

В данном примере создается индекс на столбце LastName таблицы Employees. Структура индекса позволит быстрее находить и сортировать данные по фамилии сотрудников.

Включенные и покрывающие индексы:

Некластерные индексы могут также быть расширены до покрывающих индексов, которые не только содержат ключи индекса, но и дополнительные колонки, чтобы полностью «покрыть» запрос и избежать обращения к самой таблице. Например:

```
CREATE INDEX idx_employee_lastname_includes ON Employees (LastName)
INCLUDE (FirstName, HireDate);
```

При этом запросы, которые используют как LastName, так и FirstName, могут извлекать все необходимые данные, не обращаясь к таблице, что значительно повышает производительность.

Размер некластерного индекса зависит от размера кластерного индекса, так как некластерный индекс содержит ключ кластерного индекса.

5. Специальные виды индексов (filtered, columnstore, hash, memory-optimized).

Современные СУБД предоставляют различные специальные виды индексов, которые предназначены для специфических задач и сценариев использования. Эти индексы помогают улучшить производительность запросов, оптимизировать использование памяти и ускорить доступ к данным. Рассмотрим некоторые из них.

Некластерный индекс с INCLUDE

- **INCLUDE** позволяет добавлять неключевые столбцы в некластерный индекс, что повышает его способность покрывать больше запросов.
- Индекс с неключевыми столбцами может содержать типы данных, не разрешенные для ключевых столбцов.
- **Покрывающий индекс** содержит все столбцы, необходимые для выполнения запроса, что снижает количество дисковых операций ввода-вывода и повышает производительность запроса.

Фильтруемый индекс (FILTERED)

- Оптимизированный некластерный индекс, использующий предикат фильтра для индексации части строк в таблице.
- Полезен для повышения производительности запросов к часто используемым подмножествам данных.
- Идеален для таблиц с множеством нулевых значений или столбцов, часто используемых в операторах WHERE, JOIN и ORDER BY.
- Требует меньше физического места на диске, меньше обслуживания и предоставляет более конкретные статистические метаданные.

Колоночный индекс (COLUMNSTORE)

- Данные хранятся в столбцах, что позволяет считывать только необходимые поля, минимизируя количество считываемой информации.
- Решает проблему избыточного считывания данных, особенно полезен для запросов, выбирающих одно или несколько полей из таблицы.

Хэш-индекс (HASH)

- Доступ к данным осуществляется через хэш-таблицу в памяти.
- Хэш-индекс состоит из массива указателей (хэш-контейнеров), каждый из которых содержит адрес списка ссылок на записи ключей.

- Хэш-функция применяется к ключевым столбцам индекса, результат функции определяет, к какому контейнеру относится ключ.
- Хэш-функция детерминированная, одно значение входного ключа всегда связано с одним контейнером.
- Хэш-конфликт возникает, когда несколько ключей индекса сопоставляются с одним хэш-контейнером, что может негативно влиять на операции чтения.

Оптимизированный для памяти индекс (MEMORY-OPTIMIZED)

- В таблице, оптимизированной для памяти, каждый индекс также оптимизирован для памяти.
- Строки данных хранятся не на страницах, что исключает традиционную фрагментацию и коэффициент заполнения.
- Изменения индексов в таблицах, оптимизированных для памяти, не записываются на диск, только строки данных и изменения данных записываются в журнал транзакций.
- Индексы перестраиваются при переходе базы данных в оперативный режим и создаются на основе определений индексов при восстановлении БД.
- Индексы могут быть хэш-индексами или некластерными индексами, оптимизированными для памяти.

Специальные виды индексов предоставляют расширенные возможности для оптимизации производительности баз данных в особых сценариях использования и должны выбираться с учетом специфики приложения и паттернов доступа к данным.

Усложненная теория от GPT

Понял, давайте углубимся в технические детали и принципы работы специальных видов индексов.

Некластерный индекс с INCLUDE

Принцип работы

Некластерный индекс с INCLUDE позволяет включать дополнительные столбцы, которые не являются частью ключа индекса, но могут быть полезны для выполнения запросов. Эти столбцы хранятся на листовых уровнях индекса, что позволяет избежать дополнительных обращений к таблице для получения этих данных.

Технические детали

- **Структура индекса**: Индекс организован в виде B-дерева, где листовые узлы содержат ключевые значения и указатели на строки данных. Дополнительные столбцы, указанные в INCLUDE, хранятся в листовых узлах вместе с ключевыми значениями.
- **Оптимизация запросов**: Оптимизатор запросов может использовать индекс с INCLUDE для выполнения запросов, которые требуют доступ к дополнительным столбцам, без необходимости обращения к таблице. Это снижает количество операций ввода-вывода и улучшает производительность.
- **Формат хранения**: Дополнительные столбцы хранятся в листовых узлах индекса в виде дополнительных полей, что позволяет избежать дополнительных обращений к таблице. Это особенно полезно для запросов, которые часто используют эти столбцы.

Пример

```
CREATE INDEX idx_customer_name ON customers (last_name, first_name)
INCLUDE (email, phone);
```

В этом примере индекс `idx_customer_name` включает столбцы `email` и `phone`, которые не являются частью ключа индекса, но могут быть полезны для выполнения запросов.

Фильтруемый индекс (FILTERED)

Принцип работы

Фильтруемый индекс создается с использованием предиката фильтра, который определяет, какие строки будут включены в индекс. Это позволяет индексировать только подмножество данных, что снижает размер индекса и улучшает производительность запросов.

Технические детали

- **Предикат фильтра**: Предикат фильтра определяет условие, которое должно быть выполнено для включения строки в индекс. Например, индекс может включать только строки, где значение столбца равно определенному значению.
- **Оптимизация запросов**: Оптимизатор запросов может использовать фильтруемый индекс для выполнения запросов, которые соответствуют предикату фильтра. Это позволяет избежать сканирования всей таблицы и улучшает производительность.
- **Формат хранения**: Фильтруемый индекс хранит только те строки, которые соответствуют предикату фильтра. Это позволяет уменьшить размер индекса и количество операций ввода-вывода.

Пример

```
CREATE INDEX idx_active_customers ON customers (last_name, first_name)
WHERE is_active = 1;
```

В этом примере индекс `idx_active_customers` включает только строки, где значение столбца `is_active` равно 1.

Колоночный индекс (COLUMNSTORE)

Принцип работы

Колоночный индекс хранит данные в столбцах, а не в строках. Это позволяет считывать только необходимые столбцы, минимизируя количество считываемой информации и улучшая производительность запросов, особенно для аналитических запросов.

Технические детали

- **Структура данных**: Данные хранятся в столбцах, что позволяет использовать более эффективные алгоритмы сжатия. Каждый столбец хранится отдельно и может быть сжат независимо от других столбцов.
- **Оптимизация запросов**: Оптимизатор запросов может использовать колоночный индекс для выполнения запросов, которые требуют доступ к определенным столбцам. Это позволяет избежать считывания всех строк и улучшает производительность.
- **Формат хранения**: Данные в колоночном индексе хранятся в виде сегментов, каждый из которых представляет собой сжатый блок данных. Сегменты могут быть сжаты с использованием различных алгоритмов сжатия, таких как RLE (Run-Length Encoding) или Delta Encoding.

Пример

```
CREATE COLUMNSTORE INDEX idx_sales_columnstore ON sales (product_id,
quantity, price);
```

В этом примере колоночный индекс `idx_sales_columnstore` хранит данные столбцов `product_id`, `quantity` и `price` отдельно, что позволяет эффективно выполнять аналитические запросы.

Хэш-индекс (HASH)

Принцип работы

Хэш-индекс использует хэш-таблицу для быстрого доступа к данным. Хэш-функция применяется к ключевым столбцам индекса, и результат функции определяет, к какому контейнеру относится ключ. Это позволяет быстро находить записи по ключу.

Технические детали

- **Хэш-таблица**: Хэш-таблица состоит из массива указателей (хэш-контейнеров), каждый из которых содержит адрес списка ссылок на записи ключей. Хэш-функция применяется к ключевым столбцам индекса, и результат функции определяет, к какому контейнеру относится ключ.
- **Хэш-конфликты**: Хэш-конфликты возникают, когда несколько ключей сопоставляются с одним хэш-контейнером. Это может негативно влиять на производительность, так как требуется дополнительное время для разрешения конфликтов.
- **Формат хранения**: Хэш-индекс хранит данные в виде хэш-таблицы, где каждый контейнер содержит список ссылок на записи ключей. Хэш-функция должна быть детерминированной, чтобы одно значение входного ключа всегда сопоставлялось с одним контейнером.

Пример

```
CREATE INDEX idx_customer_hash ON customers (customer_id) USING HASH;
```

В этом примере хэш-индекс `idx_customer_hash` использует хэш-таблицу для быстрого доступа к данным по столбцу `customer_id`.

Оптимизированный для памяти индекс (MEMORY-OPTIMIZED)

Принцип работы

Оптимизированные для памяти индексы хранят данные в оперативной памяти, что исключает традиционную фрагментацию и коэффициент заполнения. Изменения индексов не записываются на диск, только строки данных и изменения данных записываются в журнал транзакций.

Технические детали

- **Структура данных**: Данные хранятся в оперативной памяти в виде строк, которые не разбиваются на страницы. Это исключает традиционную фрагментацию и коэффициент заполнения.
- **Журнал транзакций**: Изменения данных записываются в журнал транзакций, что позволяет восстановить состояние базы данных в случае сбоя.

- **Перестроение индексов**: Индексы перестраиваются при переходе базы данных в оперативный режим и создаются на основе определений индексов при восстановлении БД.
- **Формат хранения**: Данные хранятся в оперативной памяти в виде строк, которые не разбиваются на страницы. Это позволяет избежать традиционной фрагментации и коэффициент заполнения.

Пример

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY NONCLUSTERED,
    first_name NVARCHAR(50),
    last_name NVARCHAR(50),
    email NVARCHAR(100)
) WITH (MEMORY_OPTIMIZED = ON);
```

В этом примере таблица `customers` оптимизирована для памяти, и все индексы также оптимизированы для памяти.

Специальные виды индексов предоставляют мощные инструменты для оптимизации производительности баз данных в различных сценариях использования. Выбор подходящего типа индекса зависит от конкретных требований приложения и паттернов доступа к данным.

6. Планы выполнения запросов

План выполнения запроса (execution plan) — это описание того, как система управления базами данных (СУБД) будет выполнять SQL-запрос. Это важный инструмент для анализа и оптимизации производительности запросов в реляционной базе данных. Понимание и работа с планами запроса позволяют разработчикам и администраторам БД находить и исправлять узкие места в производительности.

Relational Engine в Microsoft SQL Server отвечает за обработку SQL запросов, преобразуя текст запроса в результирующий набор данных через три основных этапа: **Query Parsing**, **Query Optimization** и **Query Execution**.

Query Parsing: Анализ и проверка синтаксиса SQL запроса.

Query Optimization: Оптимизатор запросов формирует план выполнения запроса, принимая решение об использовании индексов и методах сканирования. План выполнения запроса — это набор конкретных действий, выполнение которых приведет к итоговому результату.

Query Execution: Выполнение плана запроса и получение результирующего набора данных.

План выполнения запроса разделяется на две стадии:

- **Выборка результатов**: Включает вложенные циклы, слияние и хэш-соединение.
- **Сортировка и группировка, выполнение агрегаций**: Опциональная стадия.

SQL Server может выполнить запрос разными способами, используя различные алгоритмы обработки данных, такие как **NESTED LOOPS**, **HASH MATCH**, **MERGE**. Выбор алгоритма зависит от множества факторов, включая конфигурацию системы, доступные ресурсы, наличие индексов и количество данных.

План выполнения запроса можно просмотреть с помощью директив:

```
SET {SHOWPLAN_TEXT | SHOWPLAN_ALL} {ON | OFF}
GO
```

Для выполнения запросов необходимо установить значения параметров в OFF.

Предполагаемый план выполнения создается оптимизатором запросов без фактического выполнения запроса и не содержит метрик использования ресурсов. **Действительный план выполнения** создается после фактического выполнения запроса и отображает фактические метрики использования ресурсов. **Статистика активных запросов** создается в режиме реального времени и обновляется каждую секунду, позволяя анализировать процесс выполнения запроса.

Hints в MS SQL используются для навязывания решений оптимизатору, например, для использования индекса или определенного алгоритма. Примеры использования hints:

- **Использование индекса**:

```
FROM <таблица> WITH (INDEX (<имя индекса>))
```

- **Сплошное сканирование индексов**:

```
FROM <таблица> WITH (FORCE SEEK)
```

- **Использование алгоритма**:

```
OPTION (HASH GROUP)
```

План выполнения запроса используется для решения проблем с производительностью SQL инструкций, позволяя определить причины долгого выполнения запроса, такие как неиспользование индексов или неэффективные операции.

Понимание планов выполнения и их анализ — это ключ к оптимизации производительности запросов и эффективной работе с базами данных. Эти знания позволяют регулировать нагрузку на сервер и получать максимум от существующей инфраструктуры БД.

7. Статистика для оптимизации запросов

Статистика в системах управления базами данных (СУБД) играет ключевую роль в оптимизации выполнения SQL-запросов. Оптимизаторы запросов используют статистику для оценки наиболее эффективных планов выполнения, поэтому поддержание актуальности и точности статистики критически важно для производительности базы данных.

Инструменты и команды для работы с статистикой

- **SQL Server:**
  - `UPDATE STATISTICS table_name` — для ручного обновления статистики таблицы.
  - `sp_autostats` — для управления автоматическим обновлением статистики.
  - `DBCC SHOW_STATISTICS` — для отображения текущих статистических данных.
- **MySQL:**
  - Операция `ANALYZE TABLE` обновляет статистическую информацию о распределении индексов.
- **PostgreSQL:**
  - Команда `ANALYZE` обновляет статистику для оптимизации запросов.

Оптимизатор запросов использует статистику для вычисления оптимального плана выполнения запроса. **Стоимость выполнения** оценивается в количестве операций ввода-вывода из/в внешнюю память. Статистика помогает оценить **кратность** — число строк в результатах запроса.

Таблица systabsstats содержит общую статистику по таблице или индексу:

- Количество data pages таблицы или leaf страниц индекса.
- Количество строк в таблице или индексе.
- Высота индекса.
- Средняя длина строк данных таблицы или листов строк индекса.
- Количество перемещенных и удаленных строк.
- Количество пустых страниц.
- Коэффициент кластеризации.

Таблица sysstatistics содержит несколько строк для индексированных столбцов таблицы:

- Первая строка содержит базовую статистику столбцов: плотность соединений, количество дублирующих значений при соединении по столбцам, плотность аргументов поиска, количество шагов в гистограмме для столбца.

- Остальные строки содержат ячейки гистограммы, включая долю null-значений и диапазоны значений.

Пример использования статистики:

```
SELECT *
FROM student
WHERE st_city = 'СПб'
GO
```

Этот запрос вернет 41% записей, что позволяет оценить количество страниц, которые будут считаны при выполнении запроса, зная число строк в таблице, среднюю длину строки, размер страницы и степень их фрагментации.

Просмотр статистики в MS SQL реализуется с помощью команды:

```
DBCC SHOW_STATISTICS (<имя таблицы>, <имя столбца|индекса|статистики>)
```

Директива возвращает:

- **Name**: Имя объекта статистики.
- **Updated**: Дата и время последнего обновления статистики.
- **Rows**: Общее число строк в таблице при последнем обновлении статистики.
- **Rows Sampled**: Общее количество строк, выбранных для статистических вычислений.
- **Steps**: Число шагов в гистограмме (максимальное число шагов – 200).
- **Density**: Плотность.
- **Average Key Length**: Среднее число байтов на значение для всех ключевых столбцов в объекте статистики.
- **String Index**: Значение «Yes» указывает, что объект статистики содержит сводную строковую статистику.
- **Filter Expression**: Предикат для подмножества строк таблицы, включенных в объект статистики (NULL – неотфильтрованная статистика).
- **Unfiltered Rows**: Общее количество строк в таблице перед применением фильтра.
- **Persisted Sample Percent**: Процент материализованной выборки, используемый для обновлений статистики.

AUTO_CREATE_STATISTICS и **AUTO_UPDATE_STATISTICS** отвечают за автоматическое создание и обновление статистики. Они действуют по умолчанию и не рекомендуется их отключать. При их включении оптимизатор сравнивает количество измененных данных с пороговым значением и автоматически осуществляет пересчет статистики при его достижении.

Именованная статистика создается оператором:

```
CREATE STATISTICS <имя статистики>
ON {<имя таблицы>|<имя индекса>} (<столбец1>, [, …])
GO
```

Именованная статистика создается разработчиком дополнительно, если:

1. Настройщик производителя БД дает рекомендации по созданию статистики.
2. Запрос выбирает поля с сильно коррелированными данными.
3. Запрос выбирает подмножество данных.

Оператор создает статистику, содержащую задания:

- Полное/частичное сканирование таблиц.
- Запрет/разрешение пересчета статистики.
- Фильтрованная статистика.

Отфильтрованная статистика создается для определенного подмножества данных и может улучшить план выполнения запроса по сравнению со статистикой по полной таблице, повышая производительность запросов, которые выполняют выборку из четко определенных подмножеств данных.

Поддерживание актуальной и точной статистики является важнейшей задачей для обеспечения высокой производительности запросов в базе данных. Это позволяет оптимизатору запросов эффективно создавать планы выполнения и использовать доступные ресурсы наиболее рациональным образом.

8. Транзакции в базах данных, операторы языка SQL для управления транзакциями

Транзакции в базах данных — это последовательность одной или нескольких операций, которые рассматриваются как единое целое. При осуществлении транзакций сохраняются принципы атомарности, согласованности, изолированности и долговечности, известные как ACID-свойства. Это гарантирует, что данные остаются целостными и корректными в любых обстоятельствах.

Операторы SQL для управления транзакциями

1. **BEGIN TRANSACTION:**
   - `BEGIN TRANSACTION [<имя транзакции>]`
   - Инициирует новую транзакцию.
   - В некоторых базах данных используется как START TRANSACTION.
2. **COMMIT:**
   - `COMMIT [TRANSACTION [<имя транзакции>]]`
   - Фиксирует все изменения, выполненные в рамках текущей транзакции, делая их постоянными.
3. **ROLLBACK:**
   - `ROLLBACK [TRANSACTION [<имя транзакции> | <имя точки сохранения>]]`
   - Откатывает все изменения, выполненные в текущей транзакции, возвращая состояние данных к началу транзакции или к последней точке сохранения.
4. **SAVE:**
   - `SAVE TRANSACTION <имя точки сохранения>`
   - Создает точку сохранения внутри транзакции, к которой можно вернуться с помощью ROLLBACK.
5. **RELEASE SAVEPOINT:**
   - Удаляет точку сохранения, сокращая управленческую нагрузку.
6. **SET TRANSACTION:**
   - Устанавливает параметры уровня изоляции для текущей транзакции, который определяет степень видимости операций между транзакциями. Основные уровни изоляции включают:
     - **READ UNCOMMITTED:** Позволяет читать недоказанные изменения.
     - **READ COMMITTED:** Блокирует чтение изменений, которые не зафиксированы, минимально защищает от «грязного» чтения.
     - **REPEATABLE READ:** Предотвращает неповторяющееся чтение (non-repeatable read), но допускает фантомные чтения.
     - **SERIALIZABLE:** Полная изоляция; транзакции выполняются как если бы они шли последовательно.

Роль транзакций

Транзакции критически важны в приложениях, работающих с базами данных, обеспечивая целостность данных и корректность операций в условиях возможных сбоев и конкурентного доступа. Понимание и правильное управление транзакциями помогают избежать таких проблем, как потеря данных, противоречивые состояния и нарушение целостности.

Транзакция — это совокупность действий в базе данных, которая должна быть либо полностью успешно выполнена, либо полностью отклонена. Пример транзакции — перевод средств в банковской системе, где списание средств со счета A и зачисление на счет B должны быть выполнены атомарно.

Свойства транзакции (ACID):

- **Атомарность (Atomicity):** Транзакция должна быть атомарной единицей работы; либо выполняются все входящие в нее изменения данных, либо не выполняется ни одно из этих изменений.
- **Согласованность (Consistency):** По завершении транзакция должна оставить все данные в согласованном состоянии, применяя все правила для обеспечения целостности данных.
- **Изоляция (Isolation):** Изменения, выполняемые параллельными транзакциями, должны быть изолированы от изменений, выполняемых прочими параллельными транзакциями.
- **Долговечность (Durability):** После завершения полностью устойчивой транзакции произведенные ею действия занимают постоянное место в системе, даже в случае системного сбоя.

Режимы транзакций:

- **Неявные транзакции:** Транзакция начинается автоматически при выполнении очередного оператора DML.
- **Явные транзакции:** Транзакция начинается после BEGIN TRANSACTION.

```
SET IMPLICIT_TRANSACTIONS {ON | OFF}
```

Рекомендации по использованию транзакций:

- Не следует запрашивать ввод данных от пользователя во время транзакции.
- Не следует открывать транзакцию во время просмотра данных.
- Транзакцию не следует начинать, пока не завершится предварительный анализ всех данных.
- Транзакция должна быть как можно более короткой.

- Избирательно используйте более низкие уровни изоляции транзакций, такие как READ COMMITTED.
- Во время транзакции следует производить доступ к как можно меньшему объему данных, чтобы уменьшить количество блокируемых строк и снизить конкуренцию между транзакциями.

Вложенные транзакции: Транзакции могут быть вложенными, текущий уровень вложенности содержится в глобальной системной переменной @@trancount.

Состояния транзакций: Транзакция может находиться в одном из четырех возможных состояний, текущее состояние хранится в глобальной системной переменной @@transtate (в MS SQL не поддерживается).

Уровни изоляции транзакций: Использование более низких уровней изоляции, таких как READ COMMITTED, может уменьшить конкуренцию между транзакциями и повысить производительность.

9. Проблемы многопользовательского доступа к данным, их решение с помощью блокировок

Многопользовательский доступ к данным в базах данных может привести к нескольким проблемам, если не использовать правильные механизмы управления параллельностью. Блокировки — это один из ключевых механизмов, используемых для обеспечения целостности данных и управления доступом, когда несколько пользователей или процессов пытаются одновременно работать с одними и теми же данными.

Основные проблемы многопользовательского доступа

1. **"Грязное" чтение (Dirty Read):**
   - Возникает, когда транзакция читает данные, которые были изменены другой транзакцией, но еще не зафиксированы. Если вторая транзакция откатится, первое чтение будет использовать некорректные данные.
2. **Неповторяющееся чтение (Non-repeatable Read):**
   - Ситуация, при которой транзакция дважды читает одни и те же данные и получает разные результаты, потому что другая транзакция изменила данные между двумя операциями чтения.
3. **Фантомное чтение (Phantom Read):**
   - Это ситуация, когда строки, соответствующие условиям запроса, добавляются или удаляются другой транзакцией между двумя выполнениями одного и того же запроса в рамках одной и той же транзакции.

Блокировки как решение

Блокировки являются основным механизмом для управления параллельным доступом к данным. Они обеспечивают изолированность и целостность транзакций, предотвращая вышеперечисленные проблемы.

1. **Типы блокировок:**
   - **Блокировка на чтение (Shared Lock, S):** Позволяет нескольким транзакциям читать ресурс одновременно, но блокирует операции записи на этот ресурс.
   - **Блокировка на запись (Exclusive Lock, X):** Блокирует все другие транзакции от чтения или записи ресурса до освобождения блокировки.
   - **Обновляющая блокировка (Update Lock, U):** Заблокирует возможности других транзакций выполнять обновления в преддверии потенциального

обновления ресурса.

2. **Уровни изоляции транзакций:**
   - Определяют, насколько изменения, сделанные одной транзакцией, видны другим транзакциям.
     - **READ UNCOMMITTED:** Самый низкий уровень изоляции, допускает "грязное" чтение.
     - **READ COMMITTED:** Предотвращает "грязное" чтение, но допускает неповторяющиеся и фантомные чтения.
     - **REPEATABLE READ:** Предотвращает "грязное" и неповторяющееся чтение, но фантомные чтения возможны.
     - **SERIALIZABLE:** Самый строгий уровень изоляции, предотвращает все перечисленные выше проблемы.
3. **Стратегии блокировки:**
   - **Пессимистичные блокировки:** Предполагают, что конфликты будут и активно используют блокировки, чтобы предотвратить проблемы с доступом к данным.
   - **Оптимистичные блокировки:** Предполагают редкость конфликтов, выполняя проверку целостности во время фиксирования транзакции с откатом при обнаружении конфликтов.

Пример управления блокировками

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

SELECT Balance FROM Accounts WHERE AccountID = 1 WITH (UPDLOCK);

-- Some business logic or calculations

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;

-- Transfer money to another account
UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;

COMMIT;
```

В этом примере используется уровень изоляции SERIALIZABLE и установка блокировки на обновление, чтобы гарантировать, что никаких изменений в данных между шагами не произойдет, обеспечивая тем самым целостность и корректность проводимой операции.

Решение проблем

Использование блокировок является эффективным способом управления доступом к данным в условиях многопользовательского приложения. Однако злоупотребление блокировками может привести к проблемам, таким как взаимные блокировки или проблемы производительности из-за чрезмерно агрессивной блокировки. Поэтому важно тщательно подбирать уровень изоляции и стратегию блокировки в зависимости от конкретных требований и поведения приложения.

Проблемы многопользовательского доступа к данным в современных корпоративных информационных системах (КИС) включают:

1. **Потеря результатов обновления:** Параллельное выполнение транзакций может привести к некорректным результатам. Например, транзакции A (увеличивает сумму на 20) и B (списывает со счета 10) могут привести к некорректной сумме 190 вместо 210.
2. **Зависимость от незафиксированных результатов:** Транзакция может использовать незафиксированные изменения другой транзакции, что приводит к некорректным результатам. Например, транзакция A может использовать незафиксированные изменения транзакции B, что приводит к сумме 210 вместо 220.
3. **Несовместный анализ:** Транзакция может возвращать некорректные результаты из-за параллельного выполнения других транзакций. Например, транзакция A может вернуть сумму 620 вместо 570.

Решение проблем многопользовательского доступа с помощью блокировок:

- **X-Lock (Exclusive Lock):** Блокировка записи, используется для операций записи. Если на кортеж установлена X-Lock, то запросы на блокировку от других транзакций отклоняются и ставятся в очередь.
- **S-Lock (Shared Lock):** Блокировка чтения, используется для операций чтения. Если на кортеж установлена S-Lock, то запросы на S-Lock от других транзакций удовлетворяются, а запросы на X-Lock отклоняются и ставятся в очередь.

Алгоритм двухфазного блокирования:

1. Прежде чем начать работу с кортежем, транзакция должна установить блокировку.
2. Все операции блокирования должны предшествовать первой операции разблокирования.

Блокировки решают проблемы многопользовательского доступа, но создают проблему тупиков. СУБД ведет граф зависимости транзакций, показывающий, какая транзакция ожидает снятия блокировок другой транзакцией. Если в графе обнаруживается цикл, одна из транзакций приносится в жертву — принудительно откатывается и перезапускается через некоторый интервал времени или

инициируется исключительная ситуация. В качестве жертвы, как правило, выбирается транзакция, сделавшая наименьший объем изменений в БД.

Разработчик БД может повлиять на механизм обработки тупиков, задавая:

- **Приоритет сеанса:** Относительная важность текущего сеанса по отношению к другим сеансам, работающим с БД, с помощью директивы:

```
SET DEADLOCK_PRIORITY
```

- **Периодичность проверки возникновения тупика:** С помощью вызова хранимой процедуры:

```
sp_configure "deadlock checking period"
```

Практическое использование блокировок:

- СУБД используют до двух десятков видов блокировок для достижения большей производительности и минимизации вероятности появления тупиков.
- Блокировки могут применяться к объектам на различных уровнях иерархии: строке, странице, таблице.

10. Уровни изоляции транзакций

Уровни изоляции транзакций определяют, насколько одна транзакция в базе данных видит изменения, вносимые другими транзакциями. Они влияют на степень параллелизма и безопасность данных, балансируя между производительностью и целостностью. Стандартом SQL определены четыре основных уровня изоляции, каждый из которых подразумевает разную степень защиты от так называемых «аномалий» параллельного выполнения.

Основные уровни изоляции

1. **READ UNCOMMITTED (Чтение незафиксированных данных):** На самом низком уровне изоляции транзакция может читать данные, которые еще не были зафиксированы другими транзакциями.
    - **Аномалии:**
        - **Грязное чтение:** транзакция может видеть данные, которые в итоге будут отменены.
        - **Неповторяющееся и фантомное чтение также возможны.**
2. **READ COMMITTED (Чтение зафиксированных данных):** Чтение только тех данных, которые были зафиксированы в момент выполнения запроса.
    - **Аномалии:**
        - **Неповторяющееся чтение:** транзакция может получить разные результаты при повторном чтении, если другая транзакция изменила данные и зафиксировала их.
        - **Фантомное чтение возможно.**
3. **REPEATABLE READ (Повторяемое чтение):** Гарантирует, что данные, читанные транзакцией, остаются теми же при повторном чтении в одной и той же транзакции.
    - **Аномалии:**
        - **Фантомное чтение:** транзакции могут сталкиваться с новыми строками, добавленными другими транзакциями.
4. **SERIALIZABLE (Сериализуемая изоляция):** Самый высокий уровень изоляции, который делает транзакции полностью независимыми друг от друга. Каждая транзакция выполняется так, как если бы в базе данных существовала только она.
    - **Аномалии:** Полностью исключены.

Пример применения уровня изоляции

Для установки уровня изоляции в SQL можно использовать команду `SET TRANSACTION ISOLATION LEVEL`, например:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;

-- Ваши операторы SQL

COMMIT;
```

Выбор уровня изоляции

Выбор подходящего уровня изоляции зависит от приложения и его требований к согласованности данных и производительности. Например, для аналитических задач, где важна скорость, может быть достаточно уровня `READ UNCOMMITTED`, в то время как для финансовых операций часто требуется `SERIALIZABLE` для обеспечения строгой целостности данных. Важно учитывать и управлять балансом между консистентностью и конкурентоспособностью данных на этапе проектирования системы.

Уровни изоляции транзакций в стандарте языка SQL предложены для управления логикой работы блокировок и предотвращения появления тупиков. Выбор уровня изоляции транзакций позволяет разработчику БД влиять на возможность возникновения следующих ситуаций:

1. **Dirty read (грязное чтение):** Транзакция читает незафиксированные данные, измененные другой транзакцией. Например, транзакция B получает значение суммы, не соответствующее состоянию БД.
2. **Non-repeatable read (неповторяемое чтение):** Транзакция дважды читает данные и получает различные значения из-за изменений, сделанных другой транзакцией. Например, транзакция A дважды выполняет один и тот же запрос и получает различные результаты.
3. **Phantom rows (фантомные строки):** Одна транзакция отбирает набор данных запросом по некоторому условию, вторая транзакция добавляет, удаляет или изменяет данные, так что строки перестают или начинают удовлетворять условию запроса, после чего первая транзакция повторяет запрос и получает другой набор данных. Например, транзакция A дважды выполняет один и тот же запрос и получает различные наборы данных.

Уровни изоляции транзакций:

1. **Read uncommitted (уровень 0):**
   - Чтение разрешено как для незаблокированных данных, так и данных с X-блокировкой.
   - S-блокировка не устанавливается.

    - X-блокировка не удерживается (устанавливается на время физической записи).
2. **Read committed (уровень 1):**
    - Чтение разрешено для строк без X-блокировок.
    - S-блокировка устанавливается на время физического чтения.
    - X-блокировка удерживается до конца транзакции.
3. **Repeatable read (уровень 2):** S- и X-блокировки устанавливаются и удерживаются до конца транзакции в соответствии с алгоритмом двухфазной блокировки.
4. **Serializable (уровень 3):** Используются блокировки диапазонов ключей (key range lock).

Рекомендации по выбору уровня изоляции:

1. Для транзакций, затрагивающих большие объемы данных, лучше выбрать уровень 0 или 1.
2. Уровень 1 рекомендуется использовать в сочетании с курсорами, так как обеспечивает стабильность курсора без большого количества запросов на блокировку.
3. Уровень 2 следует использовать, если одна строка считывается несколько раз.

Установка уровня изоляции транзакций осуществляется директивой:

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED |
REPEATABLE READ | SERIALIZABLE }
```

Установленный уровень изоляции транзакций действует до конца сеанса или до тех пор, пока не будет изменен другой директивой. Получить значение текущего уровня изоляции транзакций можно с помощью директивы:

```
DBCC USEROPTIONS
```

или опросив глобальную системную переменную `@@isolation` (в MS SQL не поддерживается).

Задание уровня изоляции для отдельного запроса или таблицы можно осуществить с помощью табличных подсказок (hints), не изменяя уровень изоляции, установленный для сеанса.

11. Уровни схемы и виды блокировок

Блокировки — это ключевой механизм управления параллельным доступом к данным в базе данных. Они помогают предотвратить конфликтующие операции и обеспечивают соблюдение целостности данных в многопользовательских средах. Блокировки могут применяться на различных уровнях схемы базы данных и бывают нескольких видов в зависимости от целей и контекста использования.

Выбор блокировок

Уровень 1 (таблицы): Уменьшает параллелизм выполнения транзакций, но минимизирует затраты на обслуживание блокировок.
Уровень 3 (строки): Допускает максимально возможный параллелизм, но может приводить к большим накладным расходам на обслуживание блокировок.

СУБД повышает уровень блокировки до 1, если в таблице оказалось заблокировано много строк.

Схемы блокировки (Sybase):

1. **Блокировка всех страниц (allpages locking)**: Блокировка страниц данных и индекса. Снижает параллелизм из-за блокировки страниц индекса, которая может содержать 100–200 ключей.
2. **Блокировка страниц данных (data pages)**: Блокировка только страниц данных до конца транзакции и использование “защелок” на страницы индекса.
3. **Блокировка строк данных (datarow locking)**: Блокировка только строк данных до конца транзакции и использование “защелок” для страниц данных во время физической модификации страницы.

Виды блокировок:

1. **Совмещаемая блокировка (Shared)**: Используется для операций чтения (SELECT) и предотвращает изменение (UPDATE, DELETE) заблокированного ресурса. Может быть совмещена с другими совмещаемыми или блокировками обновления.
2. **Монопольная блокировка (Exclusive)**: Используется для операций изменения данных (INSERT, UPDATE, DELETE). Не может быть установлена, если существуют какие-либо другие блокировки на ресурсе.
3. **Блокировка обновления (Update)**: Ведет себя как что-то среднее между совмещаемой и монопольной блокировкой. На ресурсе может быть только одна блокировка обновления, но на этапе поиска данных она может совмещаться с другими совмещаемыми блокировками.

4. **Блокировки с намерением (Intent)**: Оптимизируют работу алгоритма установки блокировок. Блокировки с намерением устанавливаются до блокировок более низкого уровня и предотвращают получение другими транзакциями эксклюзивных блокировок.
    - **Блокировка с намерением совмещаемого доступа (IS)**
    - **Блокировка с намерением монопольного доступа (IX)**
    - **Совмещаемая блокировка с намерением монопольного доступа (SIX)**
    - **Блокировка с намерением обновления (IU)**
    - **Совмещаемая блокировка с намерением обновления (SIU)**
    - **Блокировка обновления с намерением монопольного доступа (UIX)**
5. **Блокировки схем (Schema)**: Предотвращают изменение структуры БД.
    - **Блокировка изменения схемы (Sch-M)**: Устанавливается при обновлении схемы таблицы и запрещает любой доступ к данным таблицы.
    - **Блокировка стабильности схемы (Sch-S)**: Устанавливается при выполнении запросов и предотвращает изменение объектов схемы, задействованных в запросе.
6. **Блокировка массового обновления (Bulk Update)**: Поддерживает несколько одновременных потоков массовой загрузки данных в одну и ту же таблицу и запрещает доступ к таблице любым другим процессам.
7. **Блокировка диапазона ключа (Key Range)**: Предотвращает проблему появления фантомов, блокируя диапазон выбранных строк. Реализует уровень изоляции SERIALIZABLE.

> GPT отвечает:
>
> ## Виды блокировок
>
> 1. Shared Lock (S): Разделяемая блокировка позволяет нескольким транзакциям читать ресурс одновременно, но блокирует операции записи.
> 2. Exclusive Lock (X): Исключительная блокировка не дает другим транзакциям ни читать, ни изменять заблокированный ресурс. Применяется при изменении данных.
> 3. Update Lock (U): Обновляющая блокировка, которая используется для предотвращения взаимных блокировок. Она заявляет намерение сделать блокировку эксклюзивной в будущем.
> 4. Intent Locks:
>   - Используются для сигнализации намерения установить более гранулярные блокировки. Это позволяет уменьшить дублирующие проверки при доступе к нижестоящим уровням (например, строки в таблице).
>   - Например, Intent Shared (IS) или Intent Exclusive (IX).

5. Schema Locks:
    - Блокировки схемы защищают структуру объекта базы данных (например, при выполнении ALTER TABLE).
    - Schema Modification Lock (Sch-M): Прямое изменение структуры.
    - Schema Stability Lock (Sch-S): Для операций, не изменяющих структуру, например SELECT.
6. Bulk Update Lock (BU): Особый вид блокировки, который используется при массовом обновлении данными.

Примеры использования блокировок:

- **UPDATE Счет SET Сумма = Сумма * 1,05**: Затронет все записи, будет использоваться блокировка таблицы.
- **UPDATE Счет SET Сумма = Сумма * 1,05 WHERE номер = 777**: Более целесообразна блокировка строк.

Эскалация блокировок позволяет оптимизировать работу сервера, заменяя множество низкоуровневых блокировок одной или несколькими блокировками более высокого уровня.

12. Оптимизация запросов

Оптимизация запросов в СУБД включает несколько этапов:

1. **Разбор и нормализация**: Контроль синтаксиса, проверка наличия таблиц и других объектов, используемых в запросе, проверка прав пользователя.
2. **Предварительная обработка**: Применение приемов оптимизации.
3. **Составление плана выполнения запроса**: Создание плана с наименьшей стоимостью.
4. **Компиляция плана**: Преобразование плана во внутренний язык.
5. **Выполнение и возврат результатов**: Выполнение запроса и возврат результатов пользователю.

Приемы оптимизации на этапе предварительной обработки:

1. **Преобразование инструкций в эквивалентные**:
   - **BETWEEN**: Заменяется на `<=` и `>=`.
   - **LIKE**: Если первый символ шаблона константа, заменяется на операции сравнения.

     ```
     LIKE 'sm%' -> >= 'sm' AND < 'sn'
     ```

   - **IN**: Преобразуется в последовательность операций `OR`.

     ```
     x IN (1, 2, 3) -> x = 1 OR x = 2 OR x = 3
     ```

2. **Преобразование выражений**:
   - Примеры:

     ```
     salary = 5000 * 12 -> salary = 60000
     x = sqrt(900) -> x = 30
     full_name = "Fred" + " " + "Smith" -> full_name = "Fred Smith"
     Num = 5 -> Num = 5.0
     ```

3. **Транзитивное замыкание аргументов поиска**: Позволяет дополнить запрос условиями для использования статистики.
   - Пример:

     ```
     SELECT s.st_name, d.disc_name, r.rating
     FROM student s, discipline d, result r
     WHERE d.disc_id = r.disc_id AND s.st_id = r.st_id AND d.disc_id =
     707
     ```

Добавление условия `r.disc_id = 707` позволяет использовать статистику по индексу `disc_id` в таблице `result`.

4. **Транзитивное замыкание соединений по равенству**: Увеличивает количество возможных вариантов последовательностей выполнения соединения.
   - Пример:

```
SELECT * FROM t1, t2, t3
WHERE t1.c1 = t2.c2 AND t2.c2 = t3.c3 AND t3.c3 = 1
```

   Добавление условия `t1.c1 = t3.c3` увеличивает количество возможных последовательностей.

5. **Преобразование предикатов запросов**: Вынесение общих частей условий, связанных по `OR`, и связывание их через `AND`.
   - Пример:

```
SELECT s.st_lname, d.disc_name, r.rating
FROM student s, discipline d, result r
WHERE (s.st_id = r.st_id AND r.disc_id = d.disc_id AND rating
BETWEEN 75 AND 90 AND s.st_lname = 'Petrov')
OR (s.st_id = r.st_id AND r.disc_id = d.disc_id AND rating
BETWEEN 75 AND 100 AND s.st_lname = 'Popov')
```

   Общая часть `s.st_id = r.st_id AND r.disc_id = d.disc_id` может быть вынесена и связана через `AND`.

Рекомендации по выполнению запросов:

1. Избегать использования функций и вычислений в условиях запросов, лучше их вынести в `SELECT`.
2. Избегать соединений по столбцам, для которых требуется преобразование типов.
3. Использовать как можно больше аргументов поиска.
4. Условия `>` могут работать лучше после замены их на `>=`.
   - Пример:

```
SELECT * FROM t1 WHERE c1 > 3
```

   Заменяется на:

```
SELECT * FROM t1 WHERE c1 >= 4
```

5. Запросы, реализуемые одним оператором, предпочтительнее чем двумя или более.

Прямые и отложенные обновления:

- **Прямые обновления**:
    1. **Обновления на месте**: Строки на странице не перемещаются.
    2. **Дешевые прямые обновления**: Строки перемещаются с тем же смещением на странице.
    3. **Дорогие прямые обновления**: Строки перемещаются на другую страницу.
- **Отложенные обновления**: Самые медленные, требуют записи в журнал отложенных обновлений и считывания журнала транзакций для удалений и вставок.

Рекомендации для ключей индекса:

1. Выбирать фиксированную длину для ключей индекса.
2. Если поле `varchar` близко к максимальной длине, лучше использовать `char`.
3. Неполное заполнение страниц повышает вероятность прямого обновления.

13. Определение стоимости плана выполнения запроса

Определение стоимости плана выполнения запроса (query execution plan) — это важный аспект оптимизации производительности баз данных. Каждый SQL-запрос, выполненный в системе управления базами данных (СУБД), проходит фазу планирования, где СУБД анализирует различные способы выполнения запроса и выбирает наиболее эффективный (чтобы минимизировать затраты на выполнение).

Что такое план выполнения?

План выполнения — это набор операций, которые СУБД выполняет для осуществления данного SQL-запроса. Этот план включает в себя выбор индексов, способы соединения таблиц, порядок выполнения операций и так далее.

Определение стоимости плана выполнения запроса в СУБД основывается на использовании статистики для оценки количества строк, которые будут возвращены запросом. Например, для запроса:

```
SELECT * FROM student WHERE student.st_city = 'Пятигорск'
```

Если в таблице 5000 записей и ячейка плотности гистограммы по столбцу `student_city` для значения "Пятигорск" равна 0,054, то запрос вернет 5000 * 0,054 = 270 строк.

Оценка стоимости выполнения запроса производится в условных единицах:

- Физический ввод/вывод страницы (чтение/запись на диск) — 18 у.е.
- Логический ввод/вывод (кэш) — 2 у.е.

Стоимость сканирования таблицы с блокировкой всех страниц:

```
<число страниц таблицы> * 18 + <число страниц таблицы> * 2
```

где все строки таблицы считываются в кэш, а все страницы из кэша.

Оценка стоимости использования индекса:

1. Оценивается количество строк, удовлетворяющих запросу.
2. Пересчитывается в количество страниц:
    - Для страниц данных таблицы: число строк в таблице / число страниц.
    - Для листовых страниц индекса: число строк в таблице / число листовых страниц индекса.

Стоимость выполнения точечного запроса с использованием индекса:

```
SELECT * FROM student WHERE student.st_id = 33
```

```
(<число уровней индекса> + 1) * 18 + (<число уровней индекса> + 1) * 2
```

где физическое чтение и логическое чтение из кэша.

Стоимость запроса по диапазону с использованием кластерного индекса:

```
SELECT * FROM student WHERE st_id >= 33 AND st_id < 44
```

Количество страниц данных:

```
<число подходящих строк> / <число строк на странице>
```

Стоимость:

```
<число уровней индекса> * 18 + <количество страниц данных> / <количество
страниц на операцию вв/выв> * <кластерная корректировка> * 18 +
<коэффициент заполнения страниц> + <число уровней индекса> * 2 +
<количество страниц данных> * 2
```

Соединение таблиц:
Оптимизатор выбирает последовательность соединения таблиц и алгоритм выполнения соединения:

1. **Вложенные циклы (Nested Loop)**:
   - Внешняя таблица сканируется один раз.
   - Внутренняя таблица сканируется для каждой подходящей строки внешней таблицы.
   - Стоимость:

     ```
     <стоимость доступа к Table1> + <количество подходящих строк в
     Table1> * <количество страниц в Table2 сканируемых для каждой
     строки в Table1>
     ```

   - Внешней таблицей выбирается таблица с наименьшим количеством строк, удовлетворяющих условиям запроса, или таблица, обращение к строкам которой требует наибольшее количество операций ввода/вывода.
2. **Слияние (Merge Join)**:

- Полное слияние возможно, если у обеих таблиц есть индексы по полям соединения.
- Если обе таблицы имеют кластерные индексы по полям соединения, соединение может быть выполнено следующим образом:

```
SELECT * FROM table1, table2 WHERE table1.c1 = table2.c2 AND
table1.c1 BETWEEN 100 AND 120
```

    - Указатель устанавливается на верхнюю границу диапазона с помощью индекса.
    - Сдвиг вниз для слияния записей и поиска подходящих строк.
- Стоимость полного слияния: сумма стоимости сканирования обеих таблиц.

3. **Хэширование (Hash Join)**:
    - Эффективно, если в соединении участвуют много строк и хэш-таблица помещается в ОЗУ.
    - Процесс:
        1. Сканируется первая таблица, для подходящих записей вычисляется хэш-функция.
        2. Сканируется вторая таблица, вычисляется хэш-функция и сопоставляется с вычисленной хэш-функцией первой таблицы.

Рекомендации по выполнению запросов:

1. Избегать использования функций и вычислений в условиях запросов, лучше их вынести в `SELECT`.
2. Избегать соединений по столбцам, для которых требуется преобразование типов.
3. Использовать как можно больше аргументов поиска.
4. Условия `>` могут работать лучше после замены их на `>=`.
5. Запросы, реализуемые одним оператором, предпочтительнее чем двумя или более.
6. Для ключей индекса лучше выбирать фиксированную длину.
7. Если поле `varchar` близко к максимальной длине, лучше использовать `char`.
8. Неполное заполнение страниц повышает вероятность прямого обновления.

14. Абстрактные планы запросов

Абстрактные планы запросов — это концепция, используемая для анализа и оптимизации SQL-запросов без их выполнения. Они позволяют теоретически оценить различные стратегии выполнения запроса, отображая шаги, которые система управления базами данных (СУБД) будет предпринимать для выполнения запроса. Это полезно для понимания, как базы данных выбирают наиболее оптимальные способы обработки запросов.

Инструменты и использование

1. EXPLAIN:
   - Почти каждая СУБД предоставляет команду EXPLAIN, которая генерирует абстрактный план выполнения без фактического выполнения запроса. Например:

     ```
     EXPLAIN SELECT * FROM orders WHERE customer_id = 123;
     ```

2. EXPLAIN ANALYZE:
   - В некоторых базах данных, таких как PostgreSQL, EXPLAIN ANALYZE выполняет запрос и отображает как предполагаемые, так и реальные планы выполнения, со сравнением их эффективности.
3. Графические интерфейсы:
   - Некоторые СУБД, такие как SQL Server, предлагают визуальные представления плана запроса, где разработчики могут увидеть план выполнения в графической форме.

Интерпретация абстрактных планов

1. Сканирование:
   - Table Scan: Указывает на полное сканирование таблицы, что может быть неэффективно на больших данных.
   - Index Scan: Указывает на использование индексированного доступа, что обычно быстрее.
2. Соединения:
   - Nested Loop Join: Используется для небольших наборов данных или когда один из операндов предварительно фильтрован.
   - Hash Join: Эффективен для работы с большими данными без индексов.
   - Merge Join: Эффективен, когда данные отсортированы и доступны индексы.
3. Агрегация и сортировка:

    - Операции, которые могут быть ресурсозатратными, поэтому важно минимизировать их воздействие на большие наборы данных.
4. Фильтрация:
    - Оценка, какой объем данных будет отсеян на ранних этапах плана, что может значительно повлиять на общую производительность.

Абстрактные планы запросов — это мощный инструмент для анализа и оптимизации производительности SQL, позволяющий разработчикам понимать поведение запросов и вносить изменения, чтобы улучшить общую эффективность баз данных.

Абстрактные планы запросов (АПЗ) позволяют "навязать" оптимизатору некоторые решения. АПЗ могут быть:

1. Сгенерированы сервером и сохранены в системной таблице `sysqueryplans`. Для других запросов вычисляется хэш-функция и выбирается подходящий план.
2. Указаны в разделе `plan` оператора `SELECT`, `INSERT`, `DELETE`, `UPDATE` и др.

Язык АПЗ включает операторы:

- **join**: Логический оператор соединения (не задает алгоритм).
- **nl_join**: Соединение вложенными циклами.
- **m_join**: Соединение слиянием.
- **h_join**: Соединение хешированием.
- **union**: Логический оператор объединения.
- **m_union_distinct**, **h_union_distinct**: Физические операторы, реализующие слияние/хеширование.
- **m_union_all**, **h_union_all**: При теоретико-множественном/мультимножественном объединении.
- **scan**: Логический оператор сканирования.
- **i_scan**: Физический оператор сканирования индекса.
- **t_scan**: Физический оператор сканирования таблицы.
- **m_scan**: Мультииндексное сканирование.
- **store**: Логический оператор сохранения данных во временной таблице.
- **nested**: Задает порядок выполнения вложенных запросов.

Использование логических или физических операторов позволяет получать полные или частичные АПЗ. Частичный план, как правило, указывает на использование индексов. Полный АПЗ содержит:

1. Алгоритм соединения.
2. Порядок соединения.

3. Способ сканирования индекса/таблицы.
4. Имена используемых индексов.
5. Параметры сканирования: параллелизм, размер буфера ввода/вывода, стратегия кэширования и др.

Язык АПЗ включает директивы:

- **sequence**: Группирует операторы для последовательного выполнения этапов.
- **hints**: Группирует наборы подсказок.
- **table**: Идентифицирует таблицу при использовании в запросах псевдонимов.
- **work_t**: Указывает на использование временных таблиц.

Типовые случаи использования АПЗ:

1. **Указание способа доступа**:

```
SELECT * FROM student WHERE st_lname = 'Ivanov' PLAN "(i_scan
student_name student)"
```

   План `(t_scan student)` приведет к сканированию таблицы, а план `(m_scan student)` предлагает подобрать индексы оптимизатору.
2. **Указание порядка и способа соединения**:

```
SELECT * FROM student, discipline, result
WHERE student.id = result.id AND result.disc_id = discipline.disc_id
AND student.st_lname = 'Ivanov'
PLAN "(join (join (scan student) (scan result)) scan discipline)"
```

   Требует, чтобы вначале осуществилось соединение. Алгоритм соединения и объекты сканирования не задаются, а план:

```
"(nl_join (nl_join (i_scan student_name student) (i_scan
result_91200032491 result)) (t_scan discipline))"
```

   Задает как алгоритм соединения, так и способ сканирования.
3. **Указание на использование индексов без задания последовательности соединения таблиц**:

```
"(hints (i_scan student_name student) (i_scan result_912… result)
(scan discipline))"
```

4. **Указание на использование рабочих таблиц**:

```
SELECT MAX(COUNT(*)) FROM student GROUP BY st_city
PLAN "(sequence(store worktable (scan student) – сохранение данных в
рабочую таблицу
(t_scan (work_t Worktable1)) – сканирование рабочей таблицы))"
```

Другие способы использования АПЗ:

1. Создание плана директивой `CREATE PLAN`.
2. Включение/выключение автосохранения планов выполнения запросов: `SET PLAN DUMP {ON|OFF}`.
3. Включение/выключение связывания запросов с хранимыми АПЗ: `SET PLAN LOAD {ON|OFF}`.

В случаях 2 и 3 планы сохраняются в системной таблице `sysqueryplans`. При выполнении запроса для него вычисляются значения хэш-функции, по которым извлекается план его выполнения.

Другие +/- важные вопросы

Понял, давайте углубимся в технические детали и объясним каждую характеристику и улучшение.

Из чего состоит БД на физическом уровне?

База данных на физическом уровне состоит из файлов, которые хранят данные, индексы, метаданные и журналы транзакций. Эти файлы организованы в страницы фиксированного размера (обычно 8 КБ), которые являются основными единицами хранения и управления данными. Страницы группируются в экстенты, которые, в свою очередь, объединяются в файловые группы. Файловые группы могут быть первичными (PRIMARY) или дополнительными и используются для управления физическим расположением данных на диске.

Почему БД состоит так и никак иначе?

База данных организована таким образом для обеспечения эффективного управления данными и оптимизации производительности. Страницы фиксированного размера (8 КБ) позволяют эффективно управлять памятью и минимизировать фрагментацию, что улучшает производительность операций ввода-вывода. Экстенты и файловые группы обеспечивают гибкость в управлении пространством и позволяют оптимизировать распределение данных на диске, что важно для масштабируемости и надежности системы.

Зачем нужны страницы?

Страницы являются основными единицами хранения и управления данными в базе данных. Они позволяют эффективно управлять памятью и минимизировать фрагментацию, что улучшает производительность операций ввода-вывода. Размер страницы в 8 КБ был выбран как компромисс между эффективностью использования памяти и производительностью операций ввода-вывода. Этот размер позволяет минимизировать количество операций ввода-вывода, необходимых для чтения или записи данных, и обеспечивает эффективное использование кэша.

Почему существует фрагментация памяти?

Фрагментация памяти возникает из-за неэффективного использования пространства на диске, когда данные разбросаны по разным страницам. Это происходит из-за частых операций вставки, удаления и обновления данных, которые приводят к разрывам между занятыми и свободными областями памяти. Фрагментация увеличивает количество операций ввода-вывода, необходимых для чтения или записи данных, что снижает производительность системы.

Как восстанавливается БД?

Восстановление базы данных включает процесс возвращения базы данных в консистентное состояние после сбоя. Это достигается с помощью журналов транзакций, которые записывают все изменения данных. В случае сбоя СУБД использует журналы транзакций для отката незавершенных транзакций и повторного выполнения завершенных транзакций, обеспечивая целостность данных.

Зачем нужен индекс?

Индексы используются для ускорения операций поиска данных в базе данных. Они создают дополнительные структуры данных, которые позволяют быстро находить строки, соответствующие определенным условиям. Индексы уменьшают количество операций ввода-вывода, необходимых для выполнения запросов, что значительно улучшает производительность.

Как работает индекс под капотом?

Индексы обычно организованы в виде B-деревьев или хэш-таблиц. B-деревья обеспечивают упорядоченный доступ к данным, что позволяет эффективно выполнять операции поиска, вставки и удаления. Хэш-таблицы используют хэш-функции для быстрого доступа к данным по ключу. Индексы хранят указатели на физические адреса данных, что позволяет быстро находить и извлекать строки, соответствующие запросу.

Почему поиск по индексу работает быстрее?

Поиск по индексу работает быстрее, потому что индексы создают дополнительные структуры данных, которые позволяют быстро находить строки, соответствующие определенным условиям. B-деревья обеспечивают логарифмическое время поиска, что значительно быстрее линейного поиска. Хэш-таблицы обеспечивают постоянное время поиска, что также значительно быстрее линейного поиска. Это уменьшает количество операций ввода-вывода, необходимых для выполнения запросов, что улучшает производительность.

Что делает оптимизатор запросов?

Оптимизатор запросов анализирует SQL-запросы и выбирает наиболее эффективный план выполнения. Он оценивает различные стратегии выполнения запроса, такие как использование индексов, методы сканирования и порядок выполнения операций. Оптимизатор запросов использует статистику и метаданные для оценки стоимости каждого плана и выбирает план с наименьшей стоимостью, что минимизирует затраты на выполнение запроса.

Как создаются и выполняются запросы?

Запросы создаются с использованием языка SQL и выполняются в несколько этапов. Сначала запрос проходит этап разбора, где СУБД проверяет синтаксис и семантику запроса. Затем запрос проходит этап оптимизации, где оптимизатор запросов выбирает наиболее эффективный план выполнения. После этого запрос компилируется в внутренний язык СУБД и выполняется. Результаты запроса возвращаются пользователю.

Как БД узнает о блокировках?

База данных узнает о блокировках с помощью механизма управления блокировками, который отслеживает состояние блокировок на различных уровнях (строки, страницы, таблицы). Когда транзакция запрашивает блокировку, СУБД проверяет, нет ли конфликтующих блокировок, и если конфликтов нет, устанавливает блокировку. Если конфликт существует, транзакция может быть заблокирована до тех пор, пока конфликтующая блокировка не будет снята. СУБД также использует журналы транзакций для отслеживания изменений и обеспечения целостности данных в случае сбоя.