

Оптимизация запросов

Реляционная модель

Преподаватель :

канд. тех. наук, доц. Озерова Г.П.

Методы оптимизации

Основные методы оптимизации:

1. Индексация в базах данных.

Индексы в SQL

Одним из важнейших путей достижения высокой производительности базы данных на *SQL* является использование **индексов**.

Индекс ускоряет процесс запроса, предоставляя быстрый доступ к строкам данных в таблице на основе значения в некотором столбце.

Индексы в SQL

Индекс похож на алфавитный указатель в конце книги, который позволяет быстро находить информацию в книге.

Указатель представляет собой отсортированный список ключевых слов, а рядом с ключевым словом — номера страниц, где можно найти каждое ключевое слово.

Пример алфавитного указателя по курсу

DATE	1.1.6 Выбор типов данных для полей	Теория
DATE_ADD	2.2.4 Перекрестное соединение CROSS JOIN	Теория
DATEDIFF	1.6.6 Задание. Вывести информацию о длительности командировок сотрудников	Теория
	1.6.7 Задание. Вывести информацию о командировках сотрудника(ов)	Задача
	1.6.10 Задание. Вывести сумму суточных для командировок сотрудников	Задача
	1.6.11 Задание. Вывести фамилию с инициалами и общую сумму суточных	Задача
	2.4.11 Задание. Вывести заказы, доставленные с опозданием	Задача
	3.1.5 Задание. Вывести разницу в днях между первой и последней попыткой	Задача
DAY	1.6.10 Задание. Вывести сумму суточных для командировок сотрудников	Теория
DECIMAL	1.1.6 Выбор типов данных для полей	Теория
DEGREES	1.2.6 Выборка данных, вычисляемые столбцы, математические функции	Теория
DELETE	1.5.9 Запросы на удаление	Теория
	2.3.8 Удаление записей, использование связанных таблиц	Теория
DENSE_RANK	3.5.8 Задание. Статистика по отправленным решениям обучающегося	Теория

Отличия индекса от указателя в книге

Если указатель занимает несколько страниц, то для поиска нужного слова придется последовательно листать страницы индекса.

Этот процесс можно оптимизировать, если в самом начале указателя создать еще одну страницу, содержащую алфавитный список расположения каждой буквы:

- от А до Г — стр. 121,
- от Д до Ж — стр. 122
- и т. д.

Это позволит не придется перелистывать страницы указатель, чтобы найти нужное место.

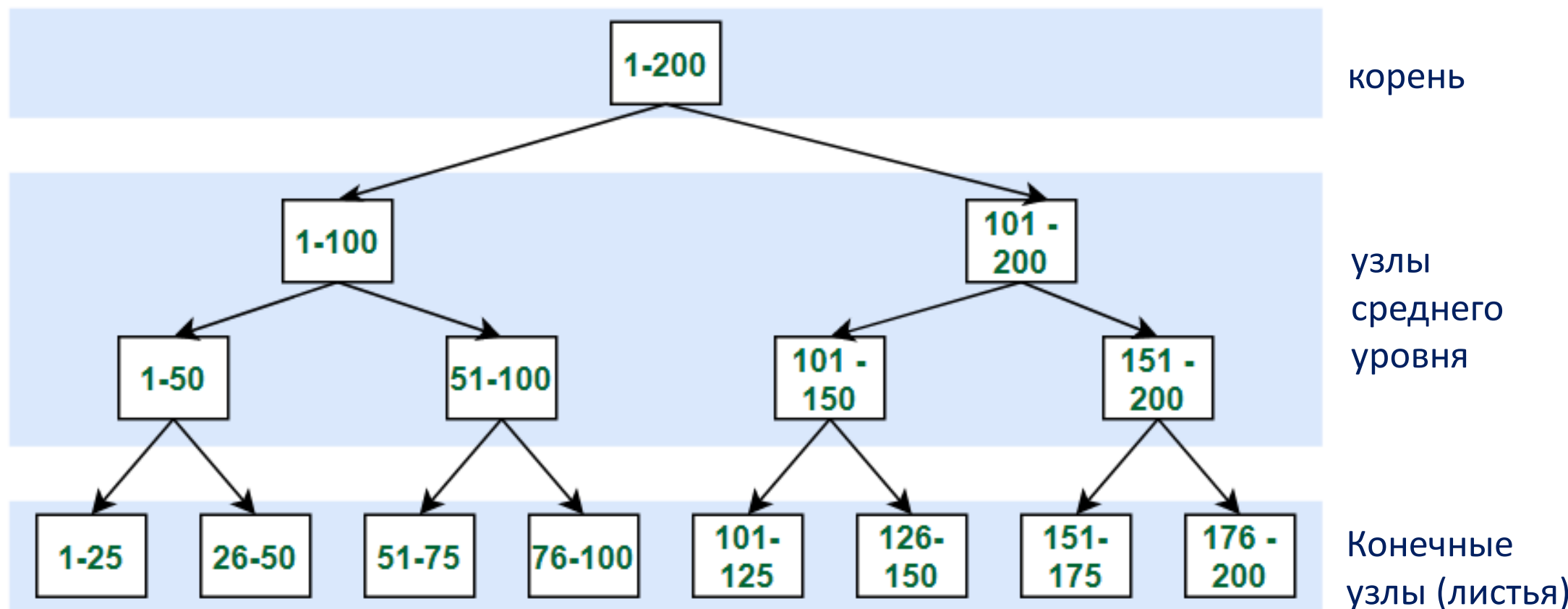
Индексы в SQL

Индекс состоит из набора страниц (узлов индекса), которые организованы в виде древовидной структуры — *сбалансированного дерева*.

Эта структура является иерархической и состоит

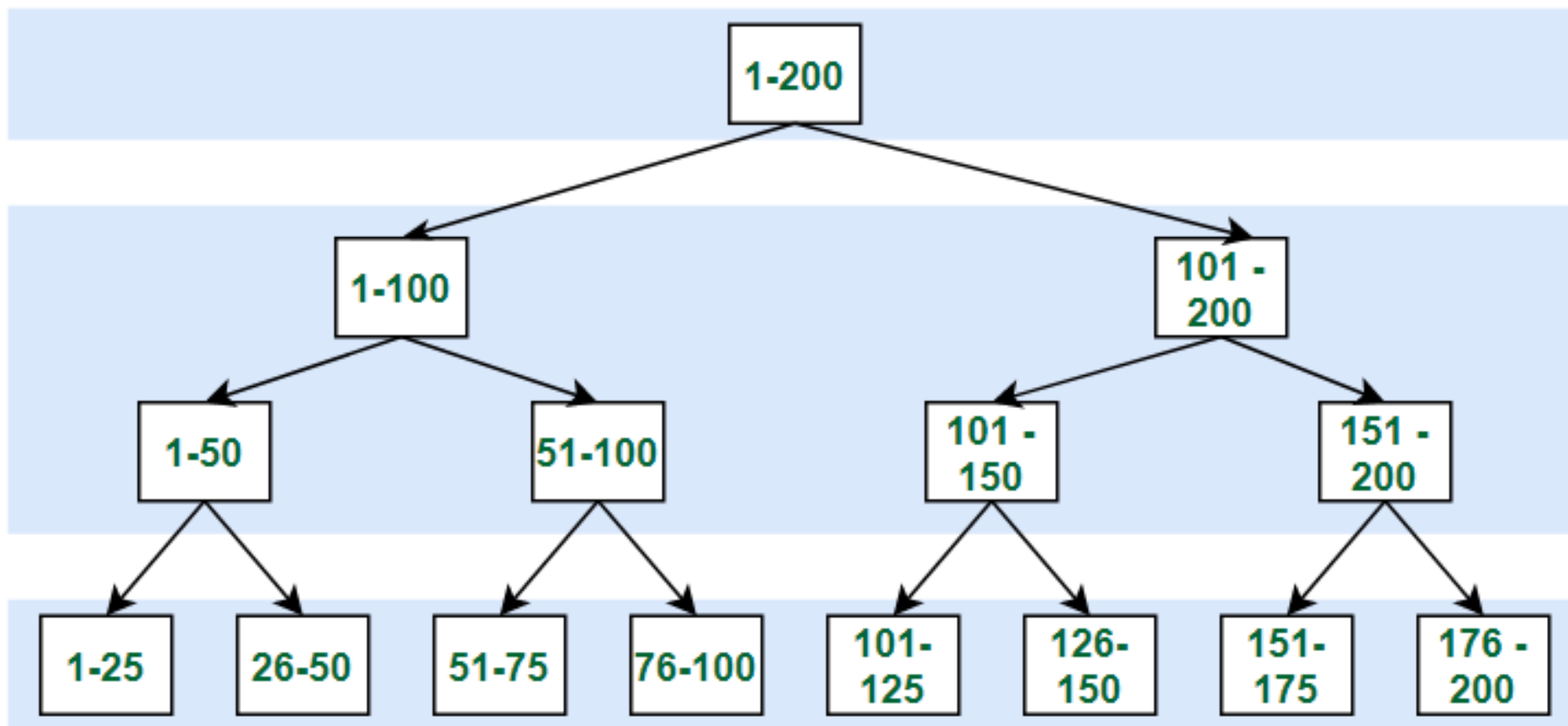
- из корневого узла;
- узлов среднего уровня;
- конечных узлов (листьев).

Индексы в SQL



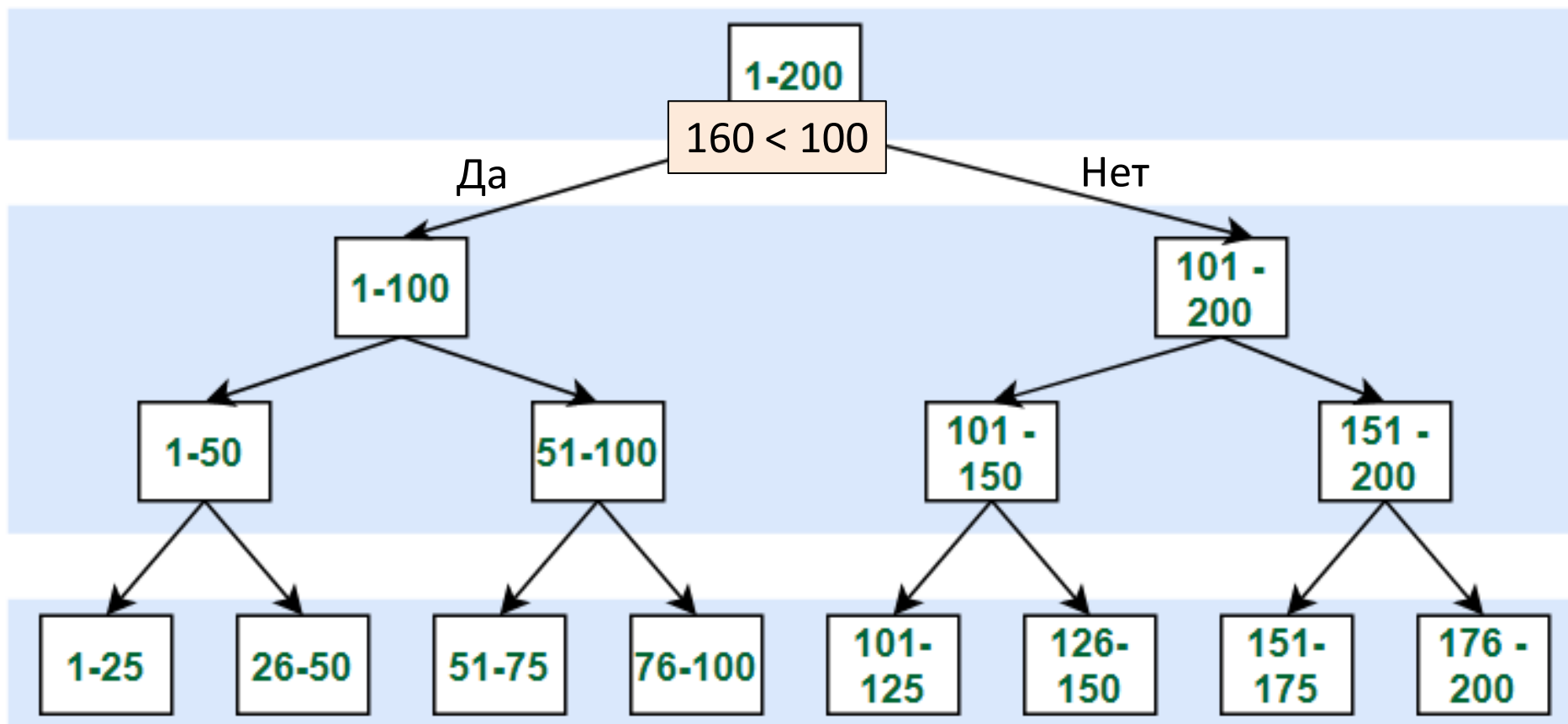
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



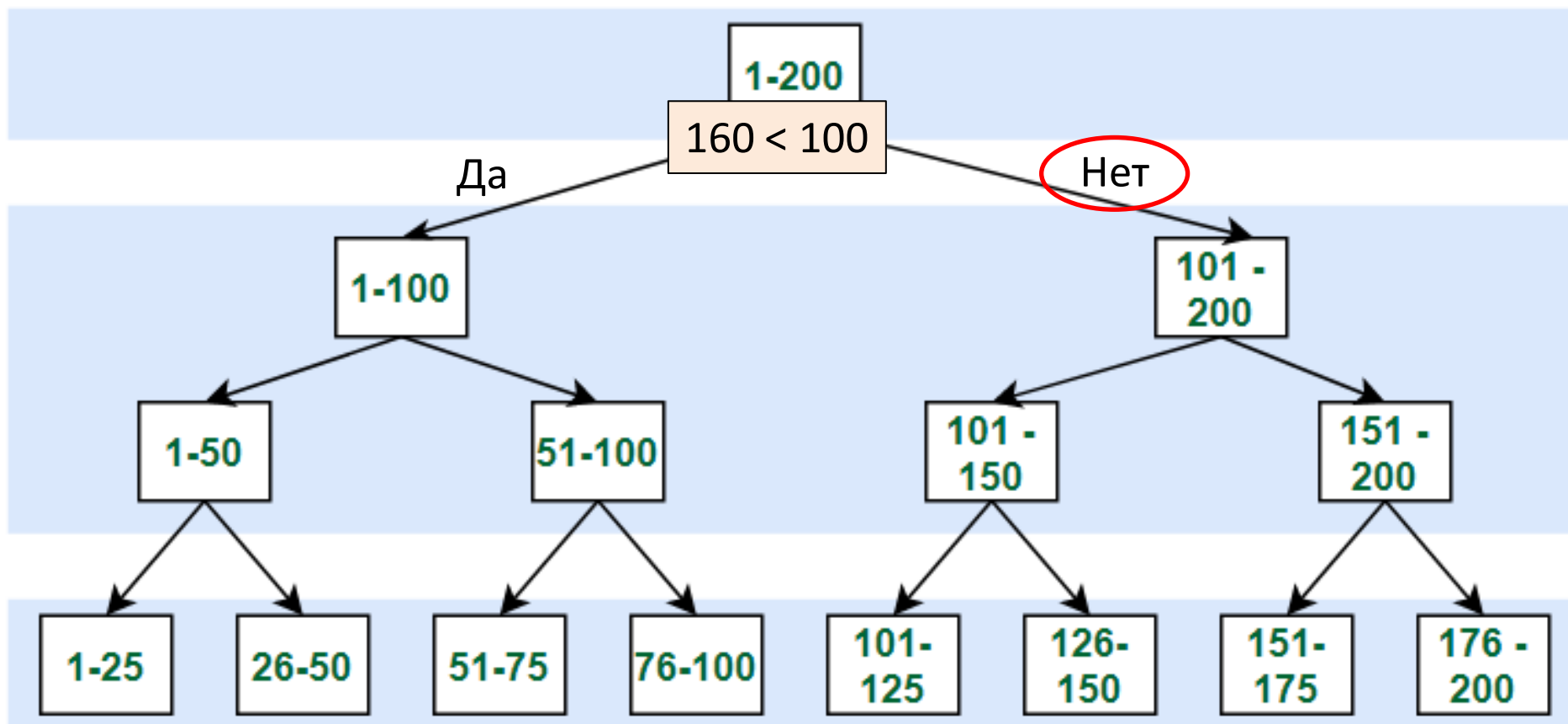
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



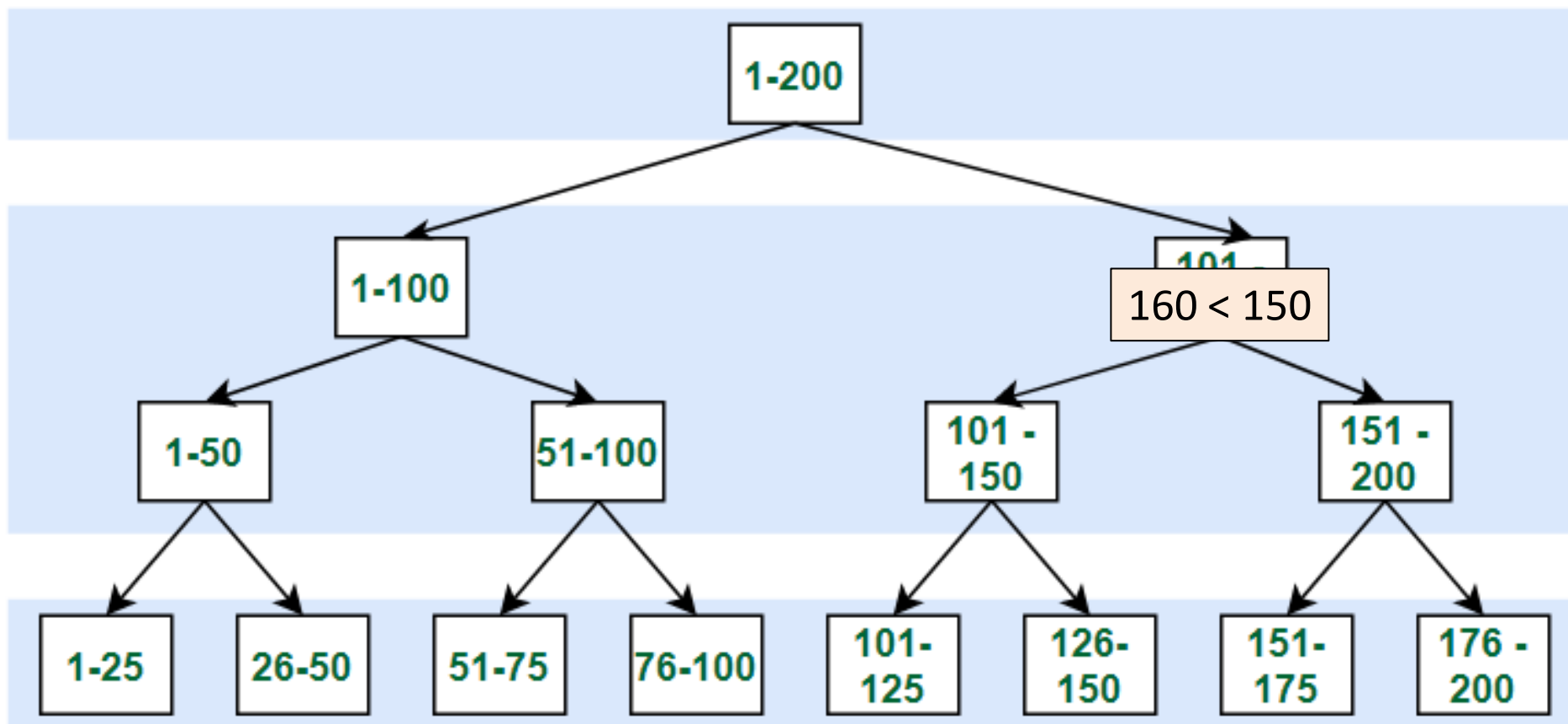
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



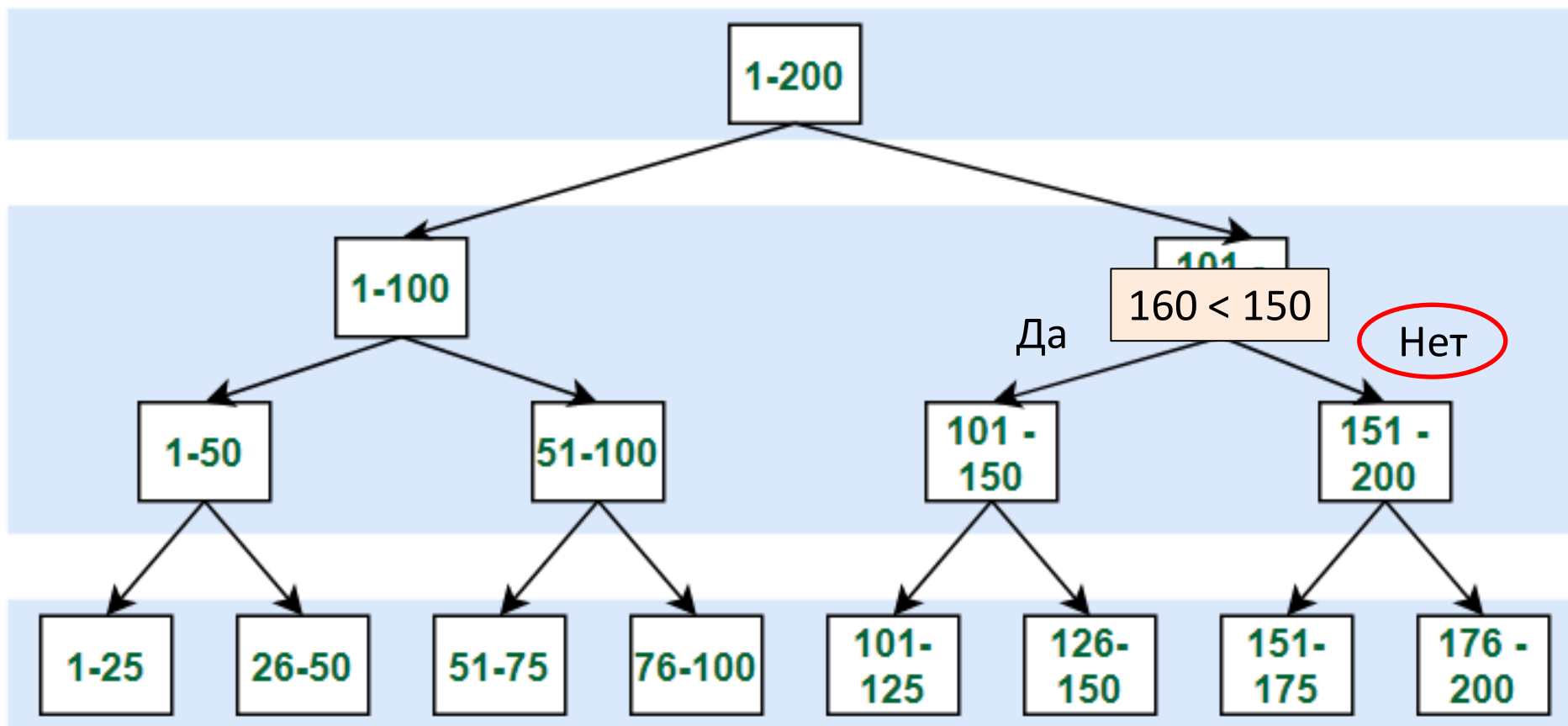
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



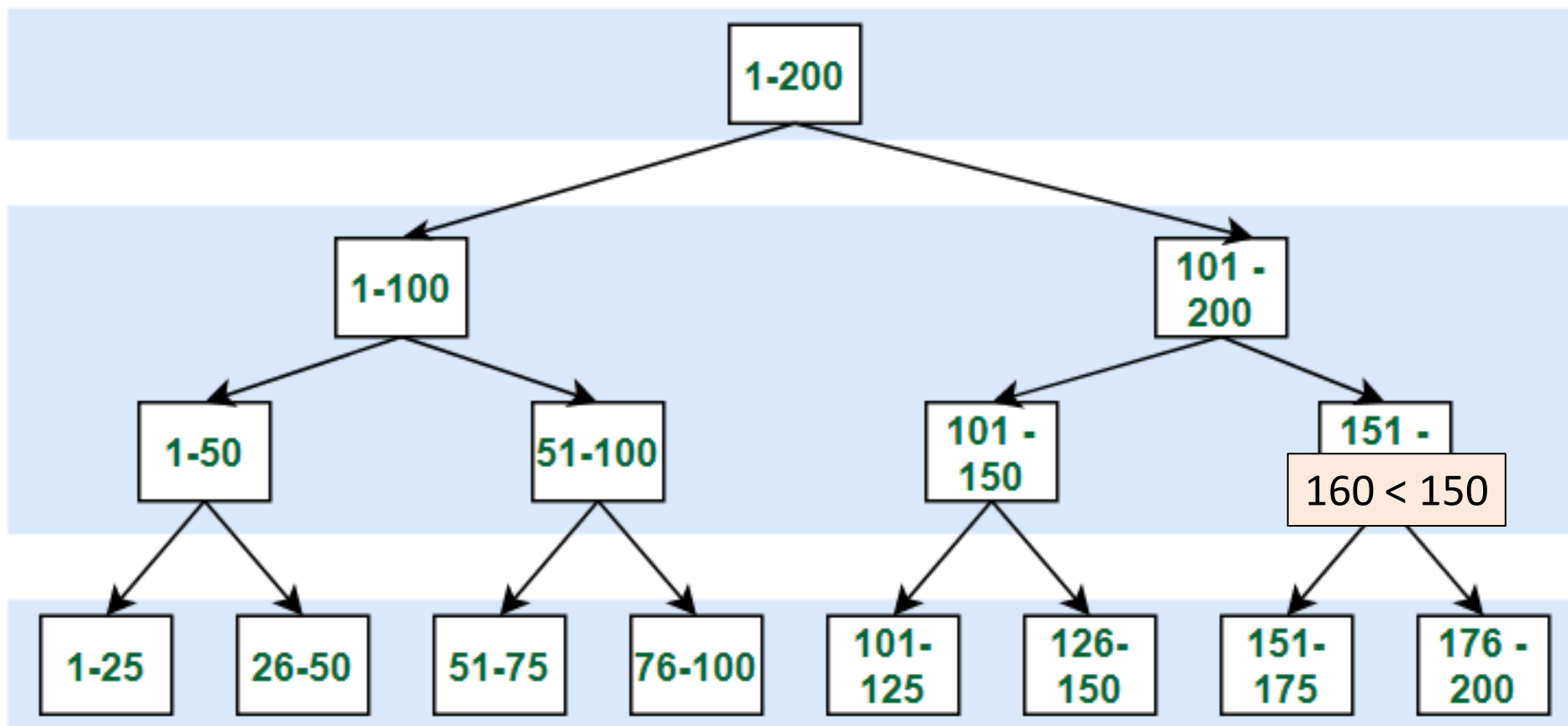
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



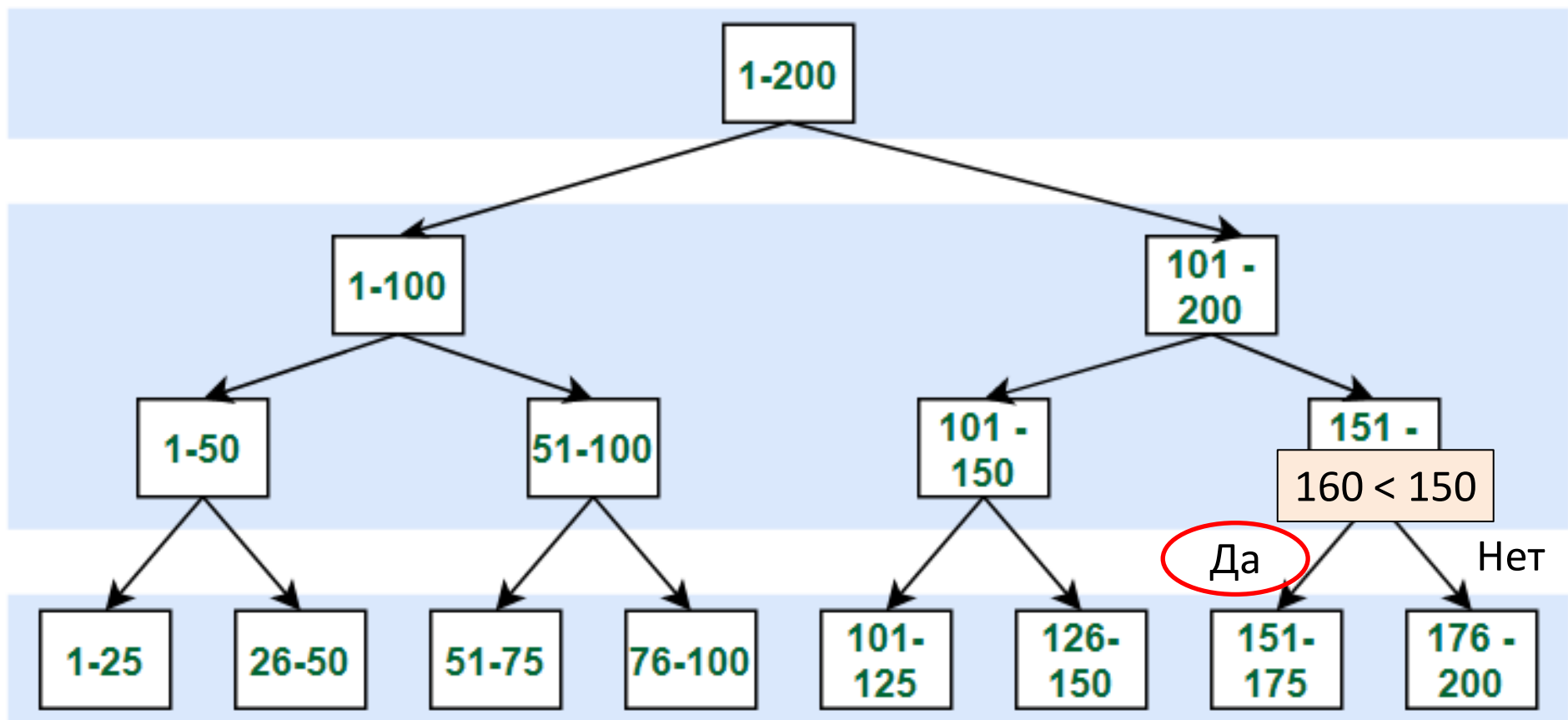
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



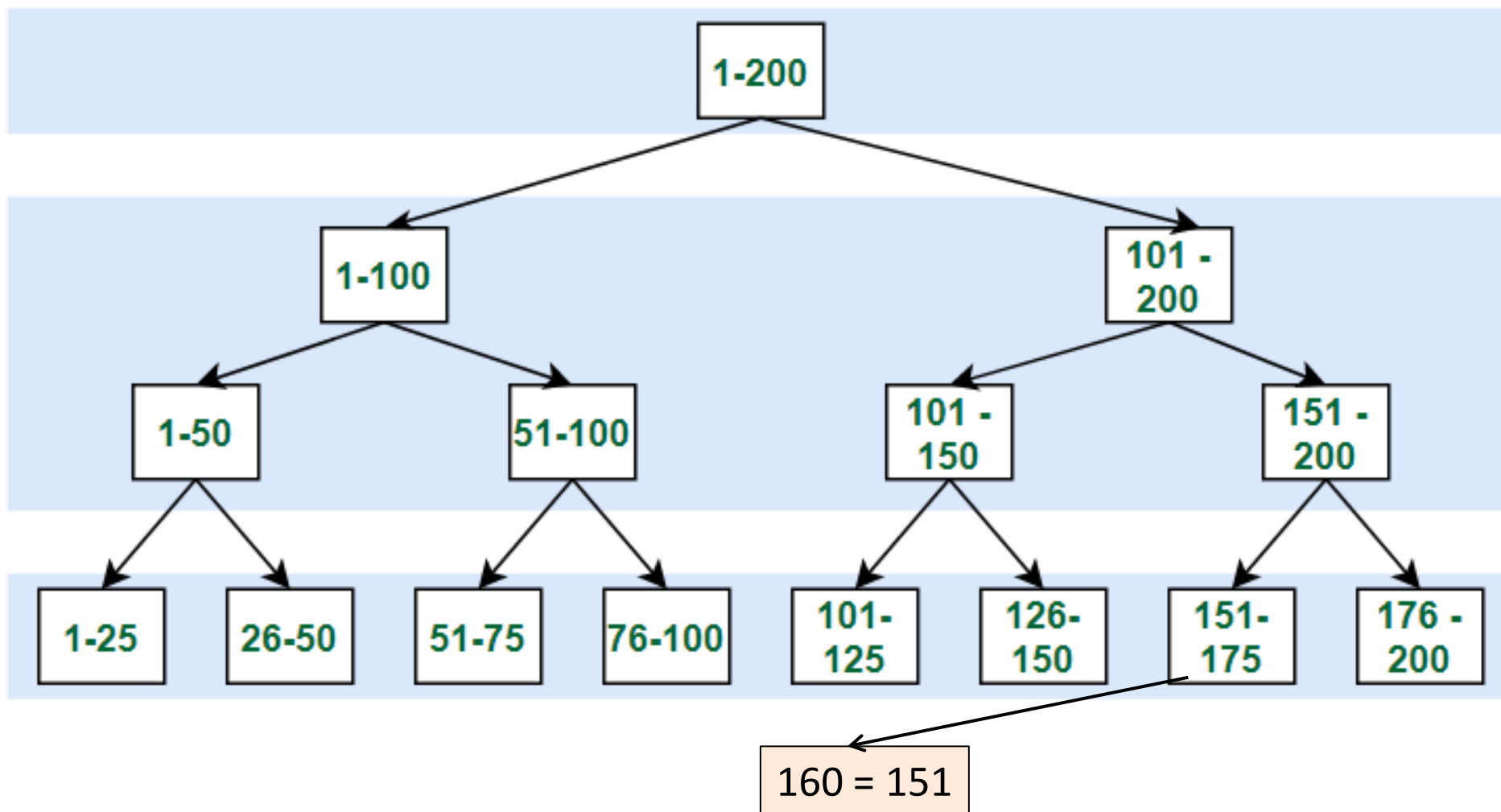
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



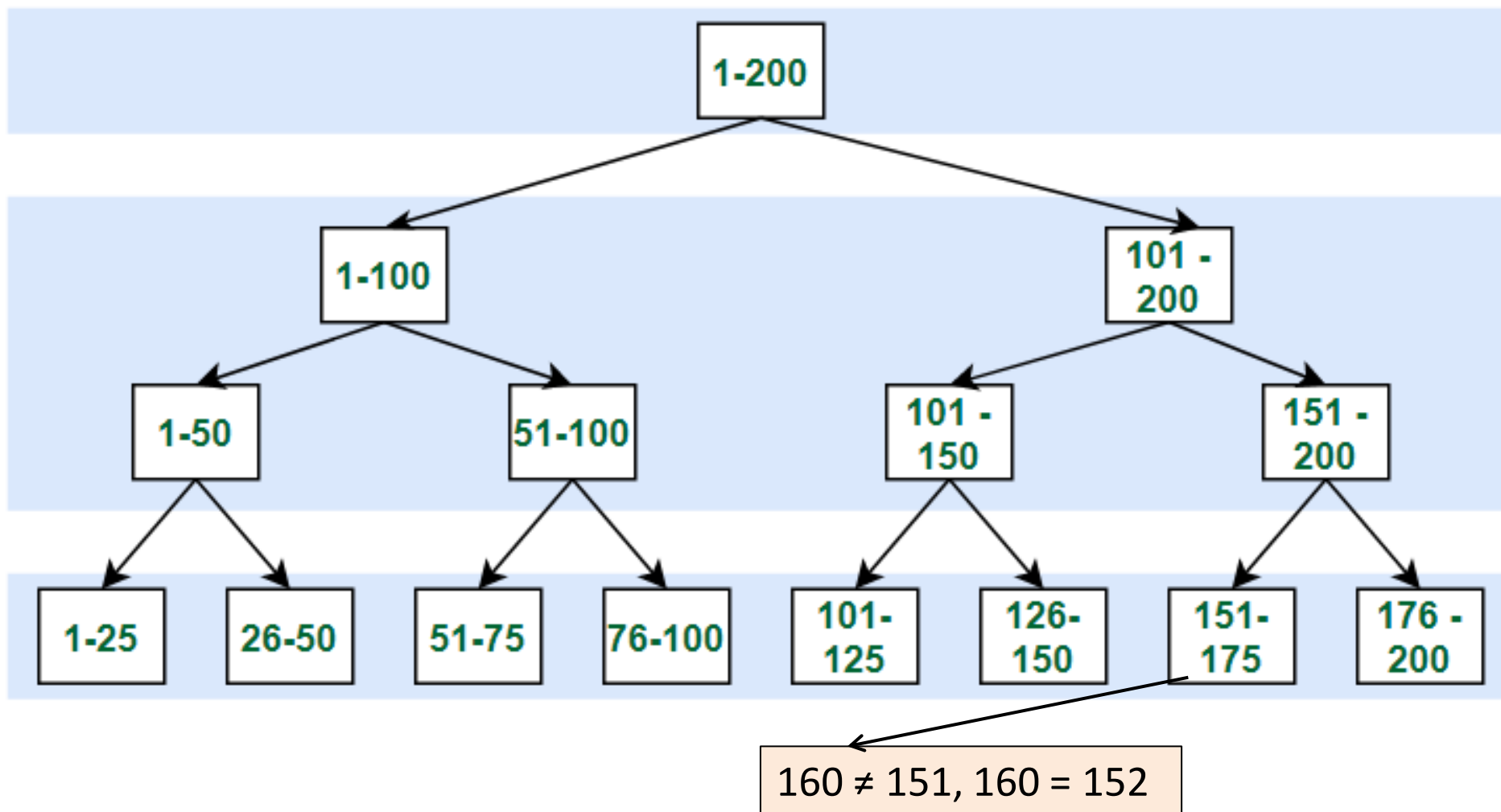
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



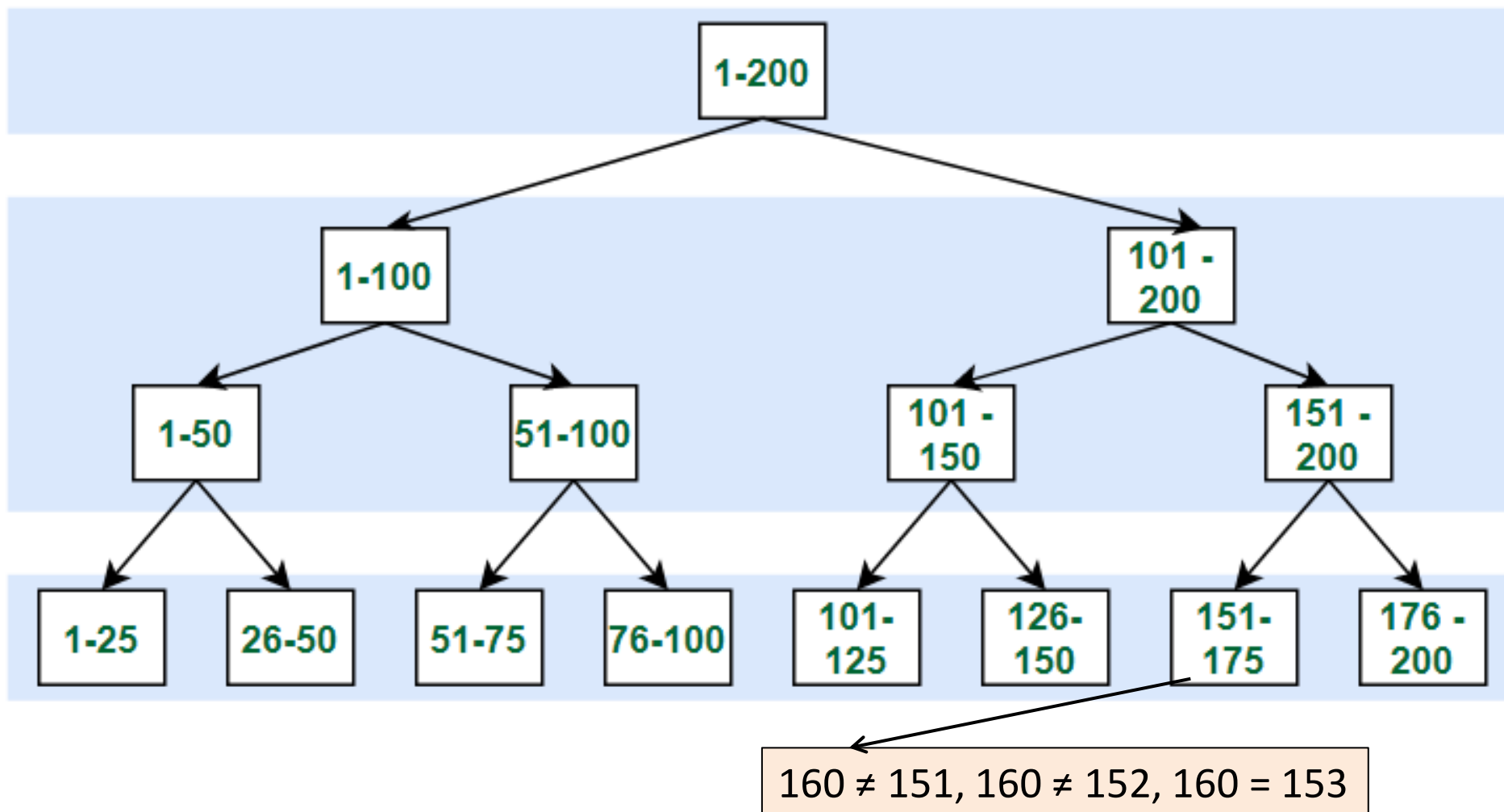
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



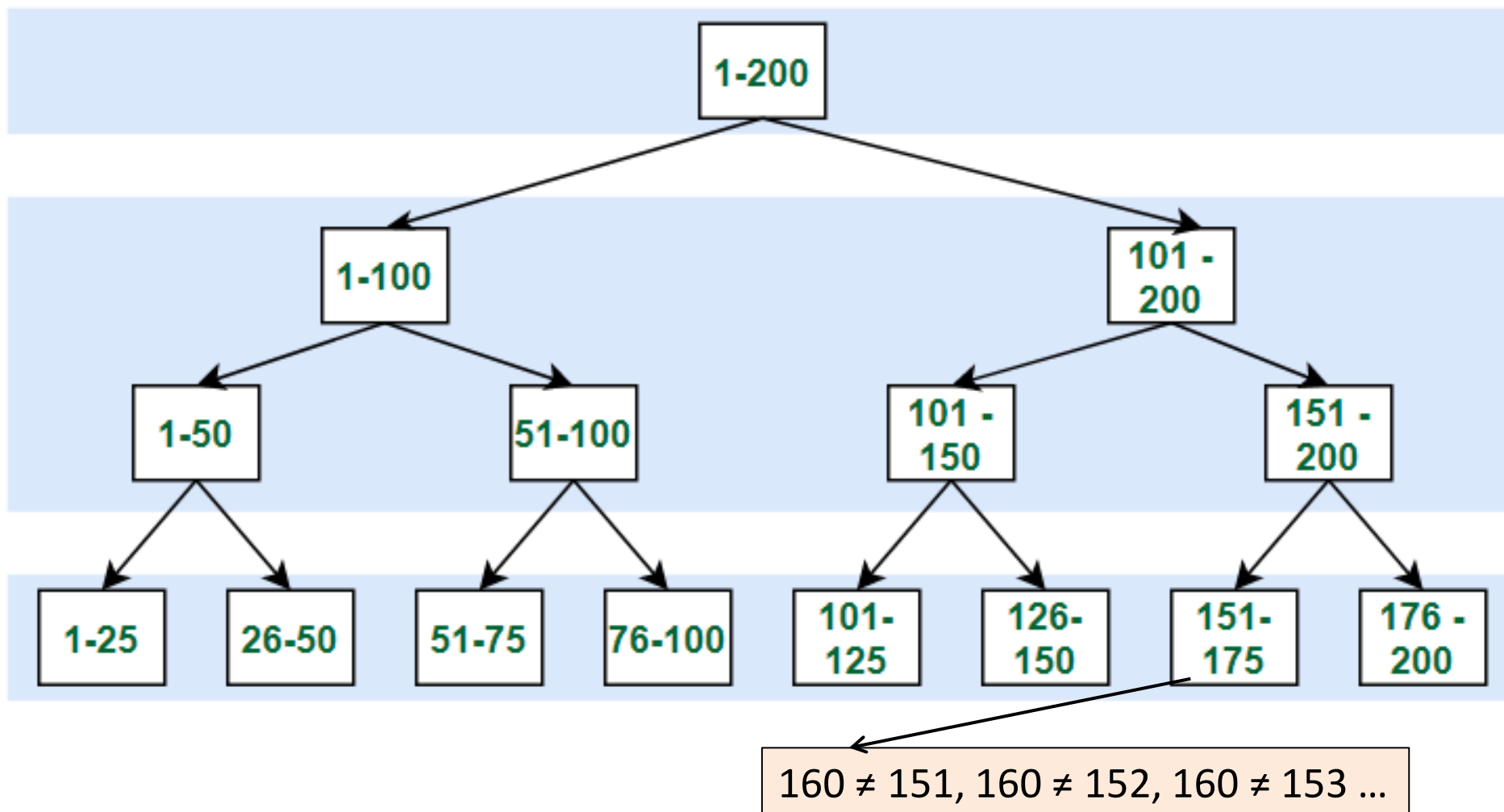
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



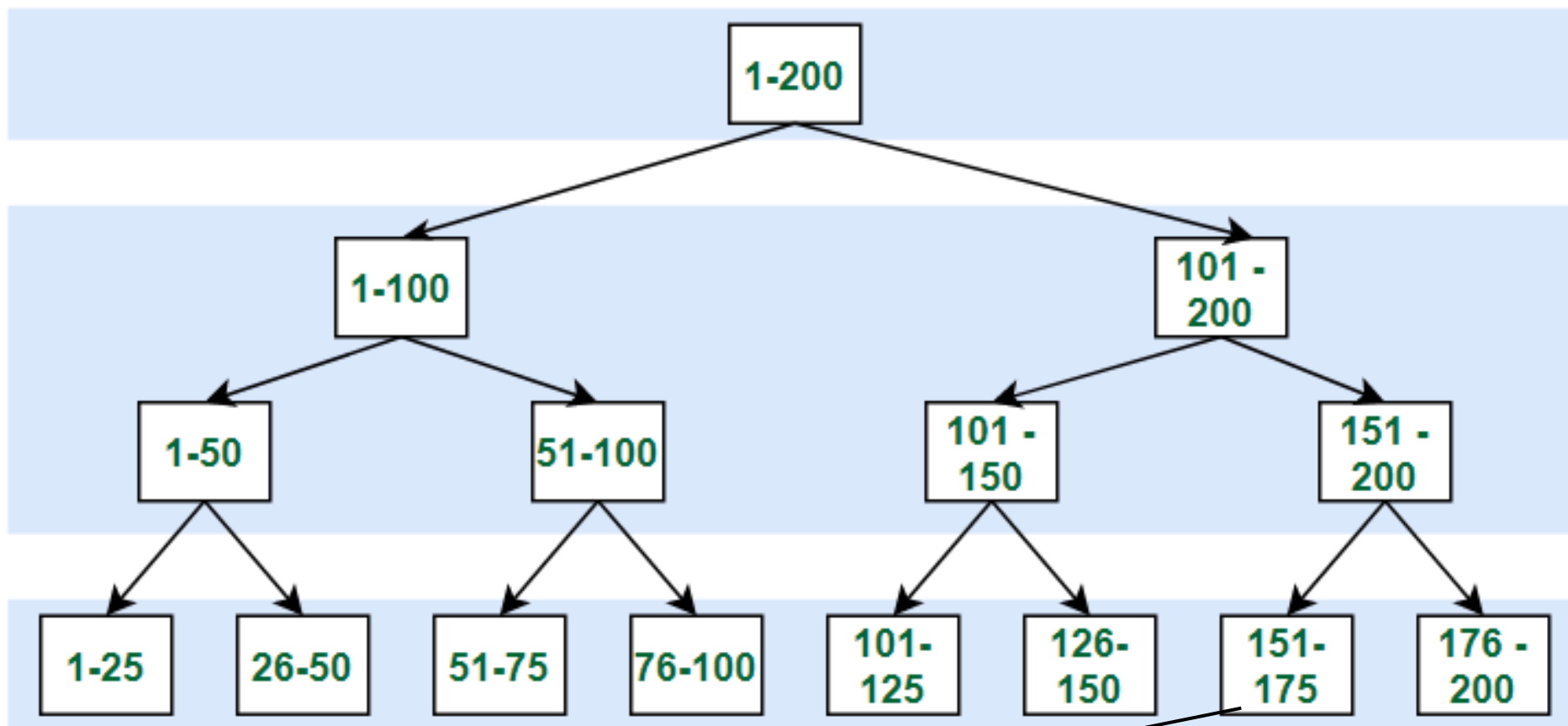
Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.



160 \neq 151, 160 \neq 152, 160 \neq 153 ... **160 = 160**

Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.

Количество операций просмотра без индекса:

- последовательный перебор 160 записей – 160.

Итого: 160.

Количество операций просмотра с индексом:

- корень – 1 ;
- узлы среднего уровня – 2;
- листья – последовательный перебор от 151 до 160 – 9.

Итого: 12.

Количество поисковых операций с использованием индекса в **13** раз меньше.

Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.

Минимальное, максимальное, среднее количество операций просмотра без индекса:

- если ищем число 1 – **1**
- если ищем число 200 – **200** .

MAX = 200

MIN = 1

AVG = 101.

Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.

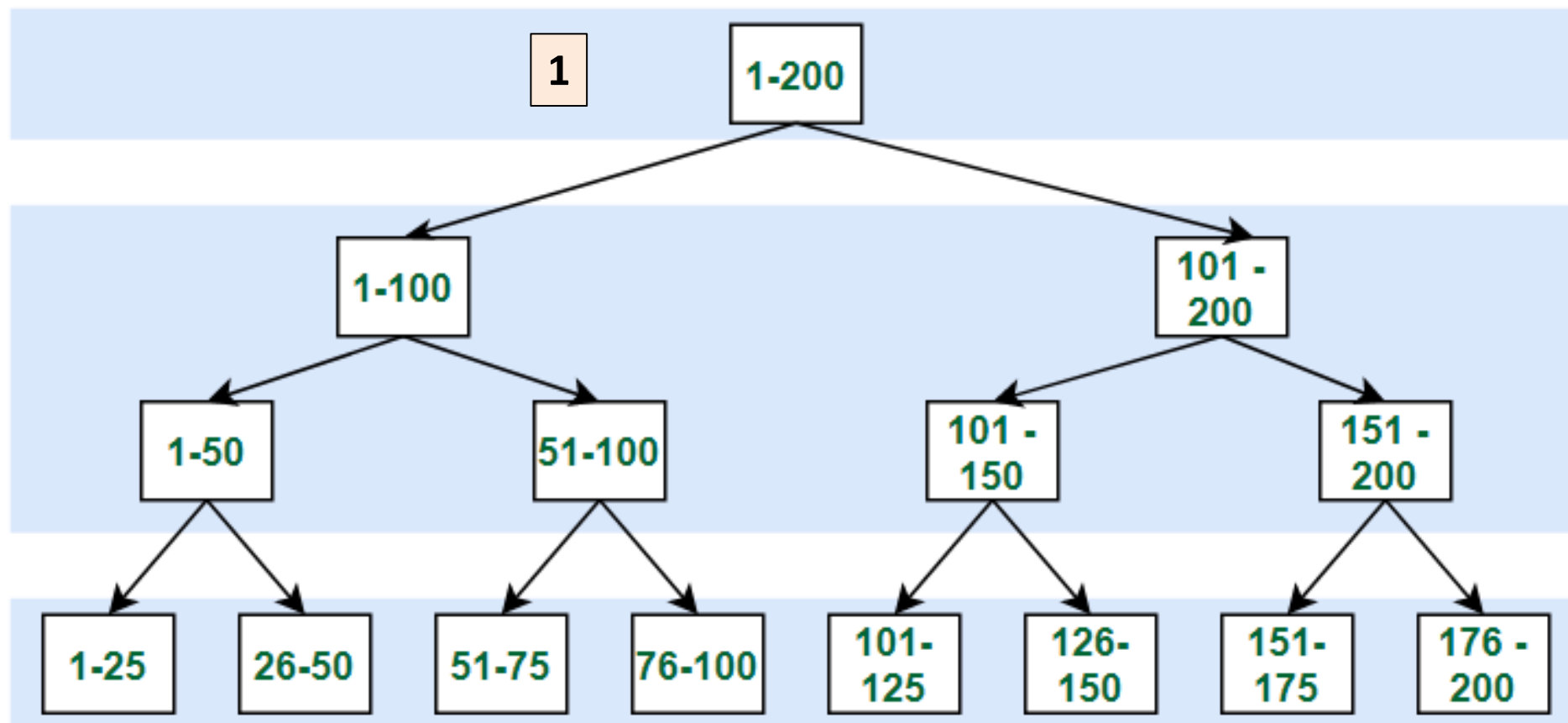
Минимальное, максимальное, среднее количество операций просмотра без индекса:

- если ищем число 1 – **1**
- если ищем число 200 – **200** .

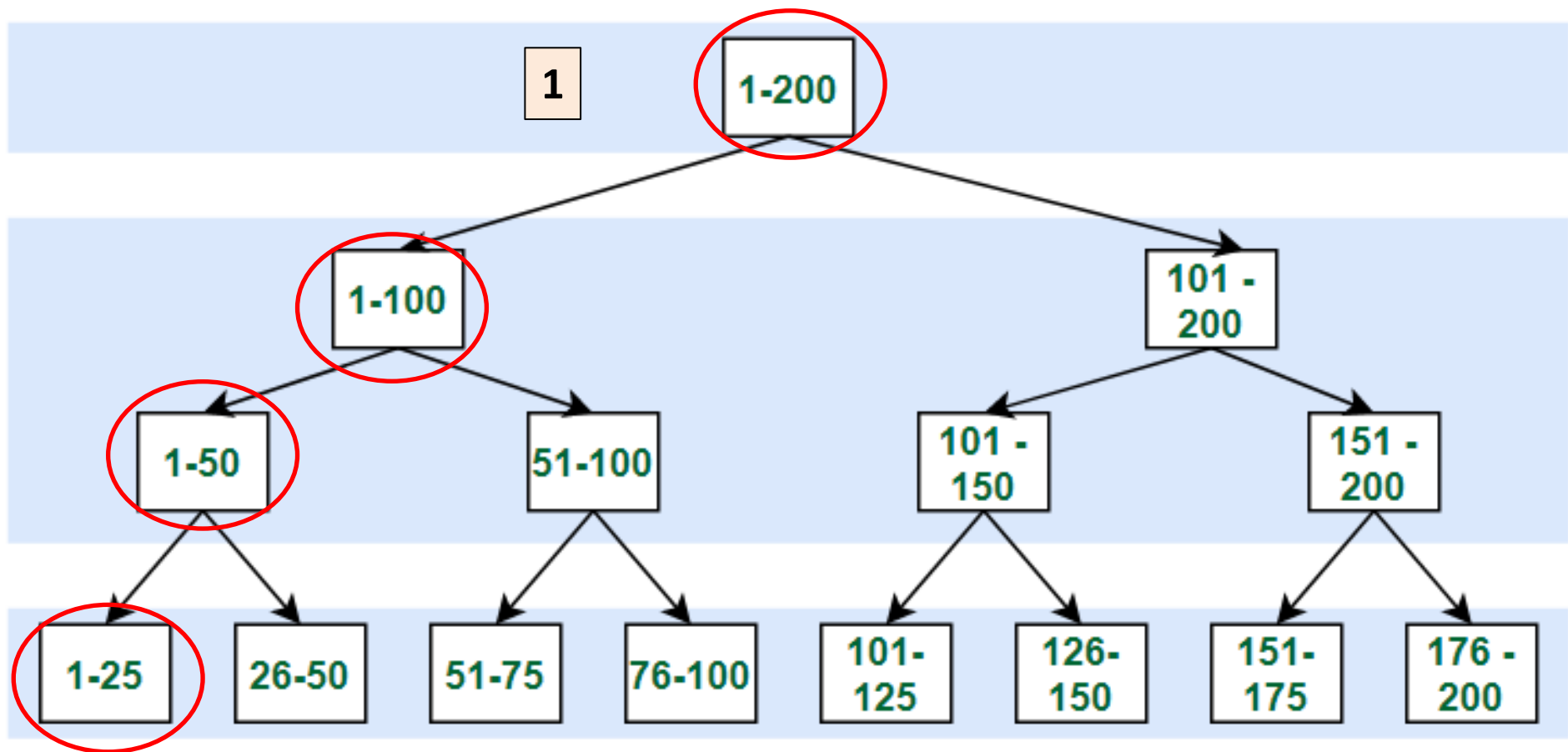
MAX = 200 MIN = 1 AVG = 101.

Минимальное , максимальное, среднее количество операций просмотра с индексом:

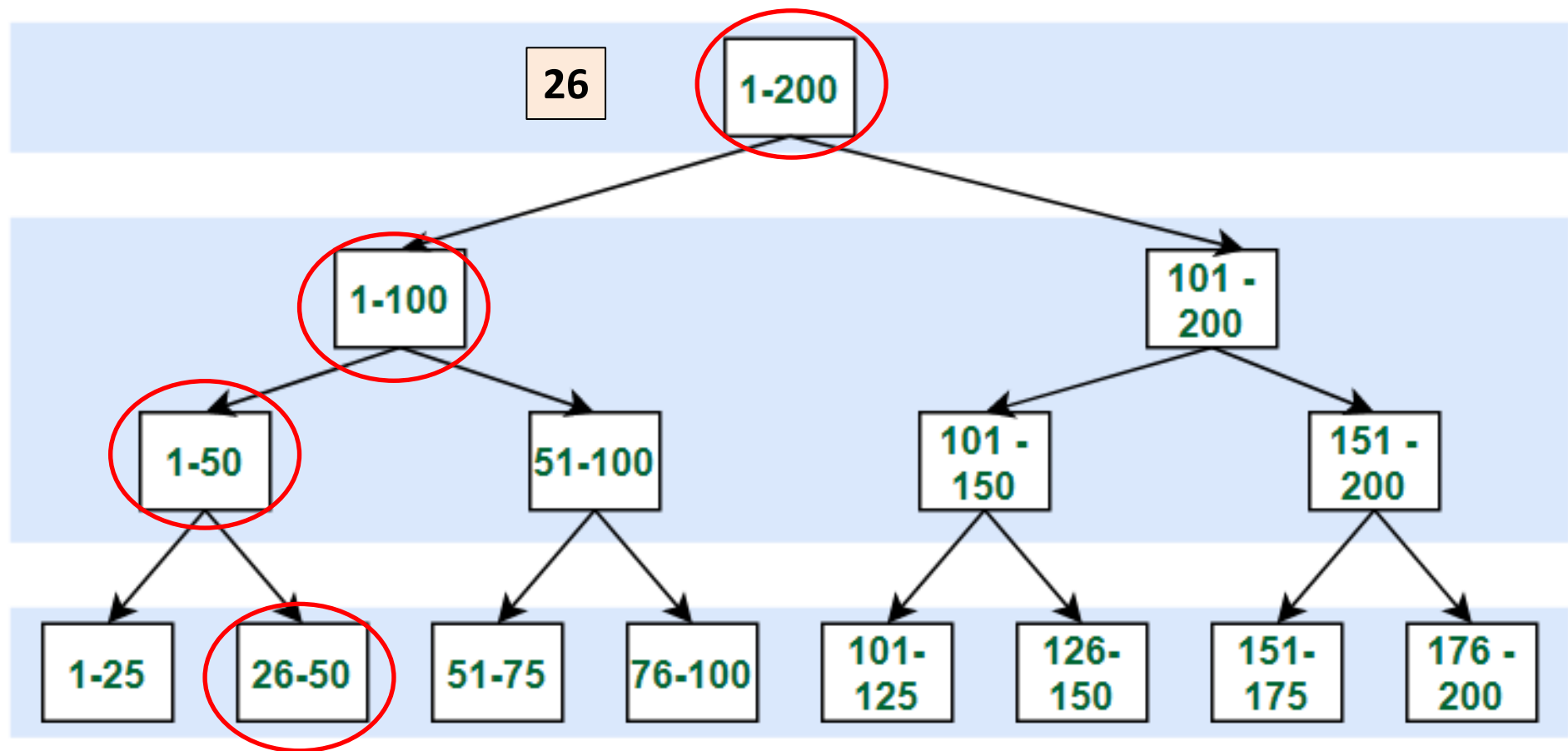
Индексы в SQL



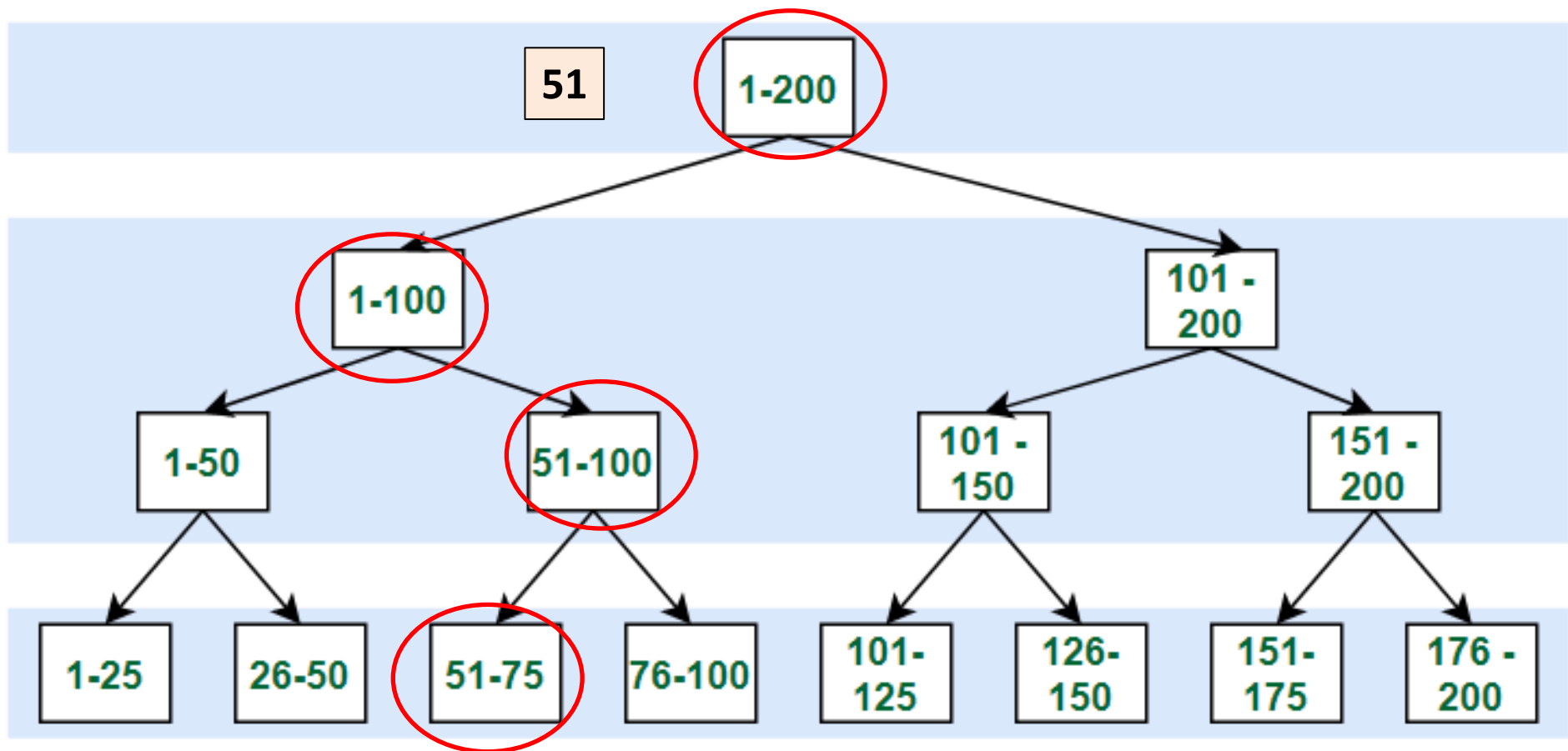
Индексы в SQL



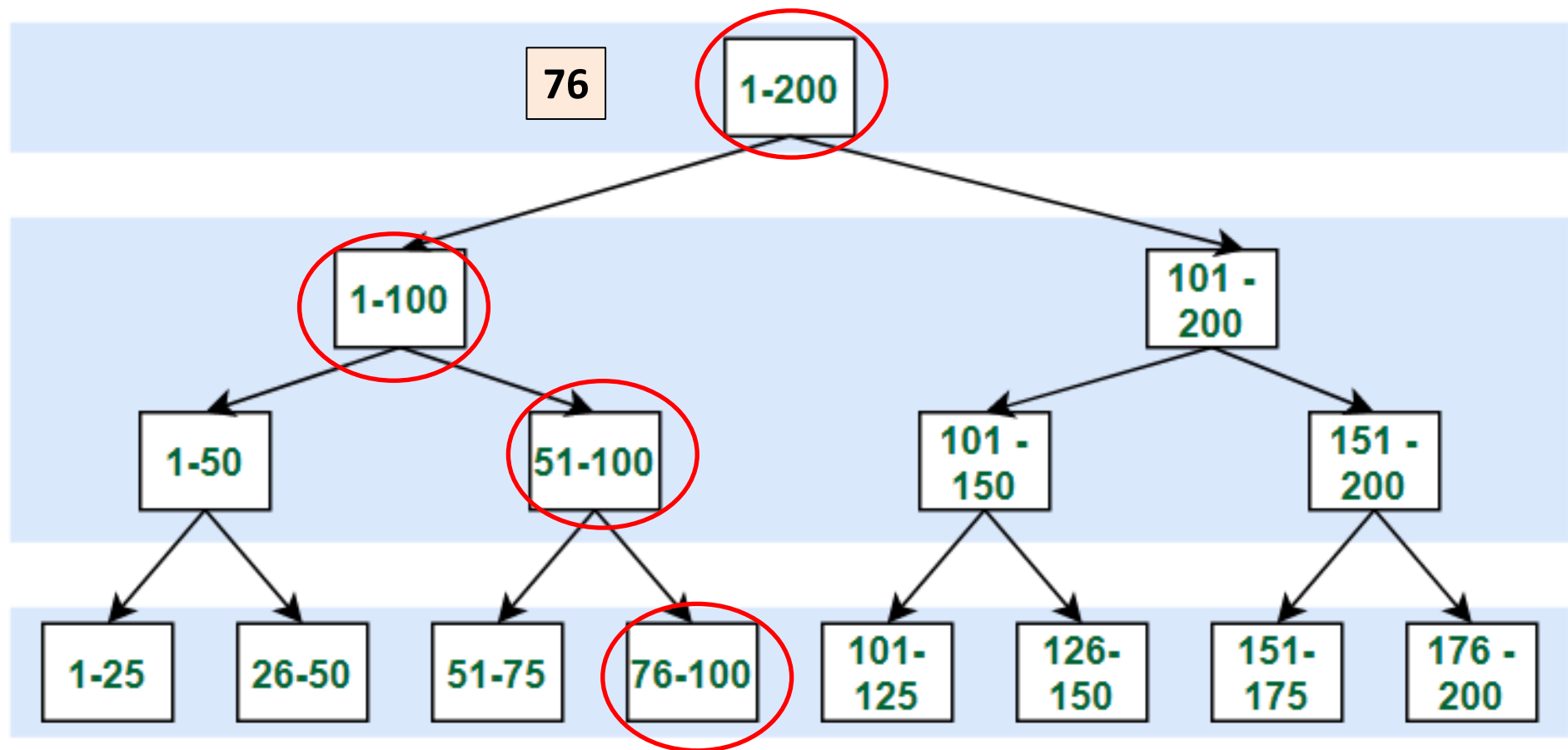
Индексы в SQL



Индексы в SQL



Индексы в SQL



Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.

Минимальное, максимальное, среднее количество операций просмотра без индекса:

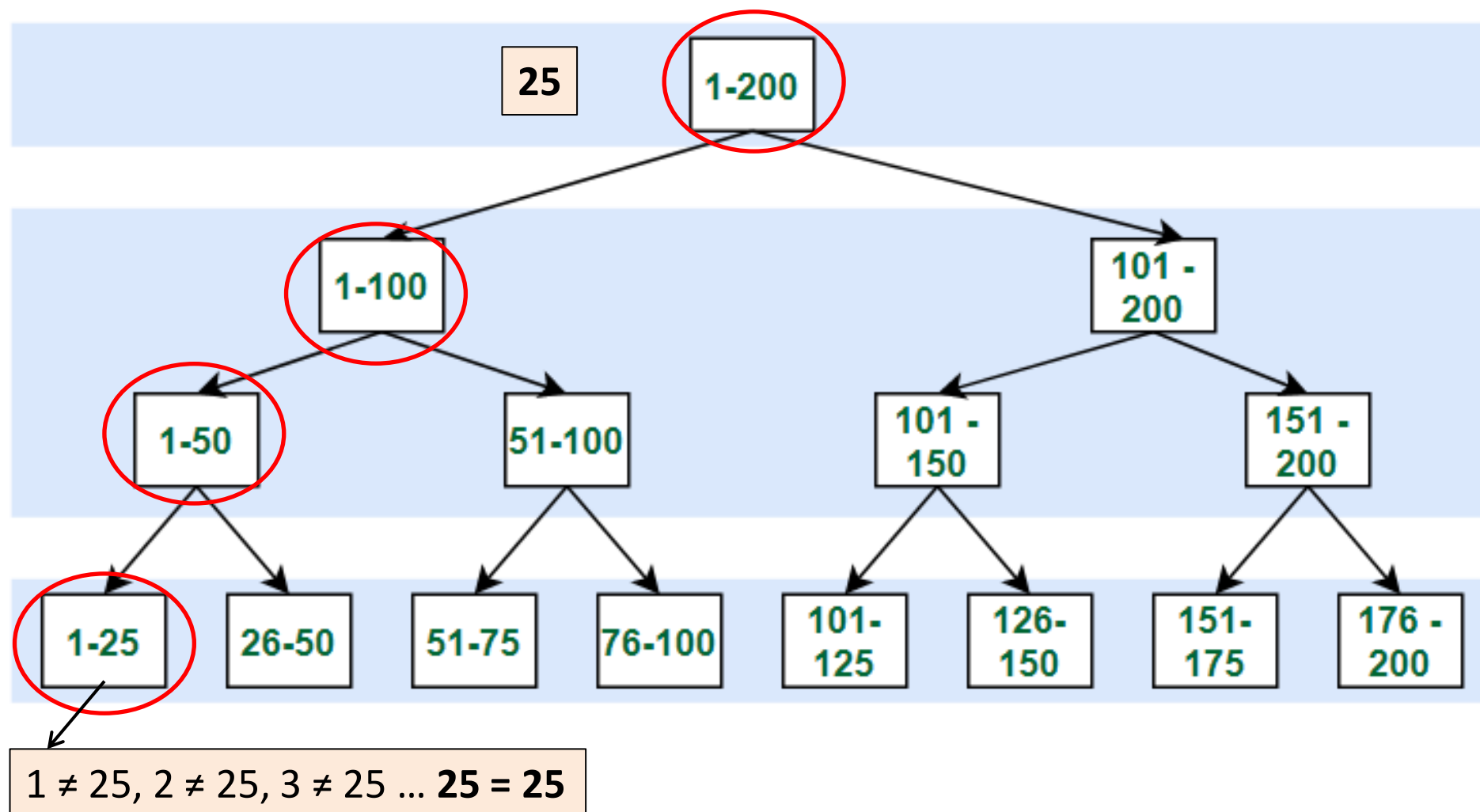
- если ищем число 1 – **1**
- если ищем число 200 – **200** .

MAX = 200 MIN = 1 AVG = 101.

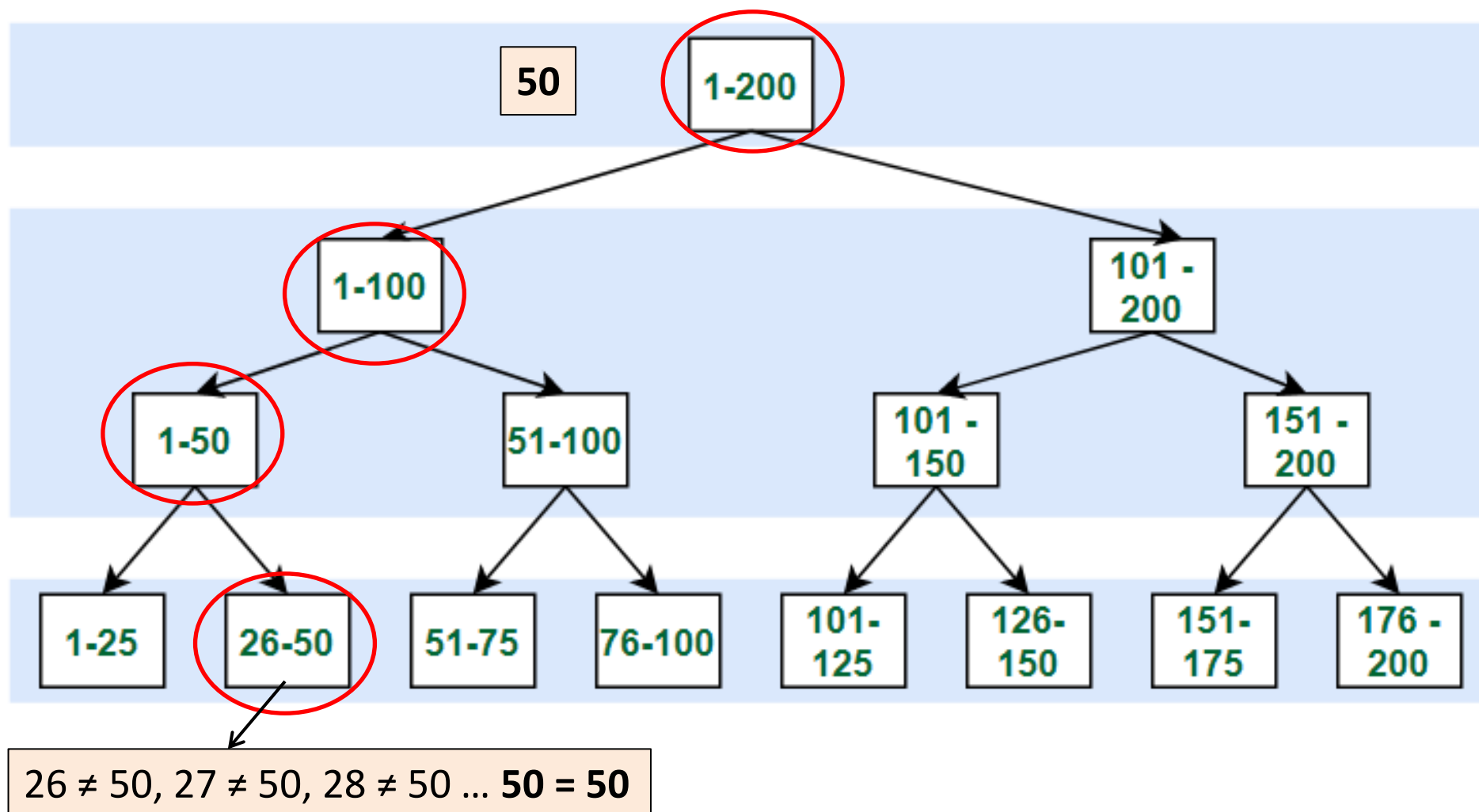
Минимальное , максимальное, среднее количество операций просмотра с индексом:

- если ищем числа 1, 26, 51, 76, 101, 126, 151, 176 – **4** ;

Индексы в SQL



Индексы в SQL



Индексы в SQL

Пример. Предположим, нам нужно найти запись 160.

Минимальное, максимальное, среднее количество операций просмотра без индекса:

- если ищем число 1 – **1**
- если ищем число 200 – **200**.

MAX = 200 MIN = 1 AVG = 101.

Минимальное , максимальное, среднее количество операций просмотра с индексом:

- если ищем числа 1, 26, 51, 76, 101, 126, 151, 176 – **4** ;
- если ищем числа 25, 50, 75, 100, 125, 150, 175, 200 – **4 + 25 = 29**.

MAX = 29 MIN = 4 AVG = 17.

Среднее количество поисковых операций с использованием индекса почти в **6** раз меньше.

Типы индексы в SQL

Структура индекса является иерархической и состоит:

- из корневого узла;
- узлов среднего уровня;
- конечных узлов (листьев).

Листья индекса могут содержать

- сами данные таблицы (**кластеризованный индекс**)
- указатель на строки с данными в таблице (**некластеризованный индекс**).

Кластеризованный индекс

Кластеризованный индекс:

- хранит реальные строки данных в листьях индекса;
- все значения отсортированы либо в порядке возрастания, либо убывания.

Таблица может иметь **только один кластеризованный индекс**.

Фактически **первичный ключ** в таблице – это кластеризованный индекс.

Таблица не имеющая кластеризованного индекса называется кучей.

Некластеризованный индекс

Некластеризованный индекс:

- листья содержат только те столбцы, по которым определен индекс, а также содержит указатель на строки с реальными данными в таблице.
- не могут быть отсортированы;
- можно создать несколько индексов (до 999).

Типы индексов

- Составной индекс
- Уникальный индекс
- Покрывающий индекс

Типы индексов

Составной индекс:

- может содержать более одного столбца;
- можно включить до 16 столбцов в индекс, но их общая длина не должна превышать 900 байтов;
- составными могут быть как кластеризованный, так и некластеризованный индексы.

Уникальный индекс

- обеспечивает уникальность каждого значения в индексируемом столбце;
- если индекс составной, то уникальность распространяется на все столбцы индекса, но не на каждый отдельный столбец;
- уникальный индекс автоматически создается при определении первичного ключа, если кластеризованный индекс не был создан ранее (в этом случае создается уникальный некластеризованный индекс по первичному ключу).

Типы индексов

Покрывающий индекс

- позволяет конкретному запросу сразу получить все необходимые данные с листьев индекса без дополнительных обращений к записям самой таблицы.

Проектирование индексов

Индексы могут улучшить производительность системы, т.к. они обеспечивают подсистему запросов быстрым путем для нахождения данных.

Но с другой стороны, если изменяются данные в таблицах, то индексы должны также быть изменены, что может значительно снизить производительность системы.

Проектирование индексов

При проектировании индексов нужно принимать во внимание несколько соображений относительно:

- базы данных;
- запросов к ней.

Проектирование индексов (база данных)

Рекомендации:

- Для таблиц которые часто обновляются используйте как можно меньше индексов.
- Если таблица содержит большое количество данных, но их изменения незначительны, тогда используйте столько индексов, сколько необходимо для улучшение производительности ваших запросов.
- Не рекомендуется использовать индексы на маленьких таблицах, т.к. возможно использование поиска по индексу может занять больше времени, чем последовательный поиск.

Проектирование индексов (база данных)

Рекомендации:

- Для индексов старайтесь использовать настолько короткие поля насколько это возможно. Наилучшим образом будет применение индекса на столбцах с уникальными значениями и не позволяющими использовать NULL (первичный ключ можно рассматривать как такой индекс).

Проектирование индексов (база данных)

Рекомендации:

- Уникальность значений в столбце влияет на производительность индекса. В общем случае, чем больше дубликатов в столбце, тем хуже работает индекс. С другой стороны, чем больше уникальных значения, тем выше работоспособность индекса, когда возможно используйте уникальный индекс.
- Для составного индекса важен порядок столбцов в индексе. Столбцы, которые часто используются в выражениях *WHERE* должны быть в индексе первыми. Последующие столбцы должны быть перечислены с учетом уникальности их значений (столбцы с самым высоким количеством уникальных значений идут первыми).

Проектирование индексов (запросы к базе)

Рекомендации:

- Старайтесь вставлять или модифицировать в одном запросе как можно больше строк, а не делать это в несколько одиночных запросов.
- Создайте индекс на столбцах которые часто используются в ваших запросах в качестве условий поиска в *WHERE* и соединения в *JOIN*.
- Использование одного составного индекса вместо нескольких односоставных для ускорения наиболее активно использующихся запросов также ускорит скорость работы с БД.
- Рассмотрите возможность индексирования столбцов, использующихся в запросах поиска строк на точное соответствие значений.

Создание индексов

```
CREATE [UNIQUE | FULLTEXT] INDEX имя_индекса  
      ON таблица (столбец [(length)] [ASC | DESC], ...  
)
```

где:

UNIQUE – индекс содержит только уникальные значения ,

FULLTEXT – содержит повторяющиеся значения (по умолчанию).

length – определяет длину символов поля для индексирования,
если параметра нет - то в индекс попадет поле целиком вне
зависимости от длины.

ст**ол**бец – поле, по которому определяется индекс.

Создание индексов (в таблице)

```
CREATE TABLE (  
    столбец ...,  
    ...  
    [UNIQUE | FULLTEXT] INDEX имя_индекса  
        (столбец [(length) [ASC | DESC]], ... )  
)
```

где:

UNIQUE – индекс содержит только уникальные значения ,

FULLTEXT – содержит повторяющиеся значения (по умолчанию).

length – определяет длину символов поля для индексирования,
если параметра нет - то в индекс попадет поле целиком вне
зависимости от длины.

столбец – поле, по которому определяется индекс.

«Измерительные» инструменты SQL

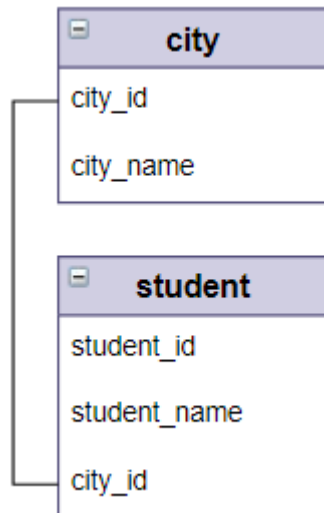
Для оптимизации запросов SQL необходимы инструменты, которые позволят оценить эффективность того или иного запроса, а затем его оптимизировать:

- **измерение времени выполнения запроса;**
- **оценка плана выполнения запроса.**

База данных для анализа

В базе данных хранится информация о городах и студентах, для каждого известно в каком городе он живет.

Логическая схема базы данных:



База данных для анализа

```
CREATE TABLE city(  
    city_id INT,  
    city_name char(20)  
);
```

city – наполнение 1000 записей

```
CREATE TABLE student(  
    student_id INT,  
    student_name CHAR(40),  
    city_id INT  
);
```

student – наполнение 10145 записей

Измерение времени выполнения запроса

Чтобы выполнить оценку используемых ресурсов запроса, используется переменная сеанса профилирования, значение по умолчанию которой равно 0 (OFF).

Чтобы начать оценку, необходимо переменной присвоить значение 1 (ON) :

```
SET profiling = 1;
```

Затем перечисляются запросы, оценку которых Вы хотите выполнить.

```
SELECT ...
```

И для просмотра результата используется оператор:

```
SHOW PROFILE[S] ...;
```

Измерение времени выполнения запроса

Рассмотрим семантику оператора **SHOW PROFILE[S]** на следующих примерах:

Запрос 1

```
SELECT COUNT(*)  
FROM student;
```

Запрос 2

```
SELECT COUNT(*)  
FROM student  
WHERE city_id = 113;
```

Измерение времени выполнения запроса

SHOW PROFILES

- показывает общую информацию (продолжительность и текст) последних запросов, отправленных на сервер (по умолчанию – 15, максимальное значение 100)

Измерение времени выполнения запроса

Эксперимент. Создадим блок из 7 одинаковых запросов, запустим его и оценим результат

```
SET profiling=1;  
запрос;  
запрос;  
запрос;  
запрос;  
запрос;  
запрос;  
запрос;  
SHOW PROFILES
```

Измерение времени выполнения запроса

Запрос 1

```
SELECT COUNT(*)  
FROM student;
```

Query_ID	Duration	Query
1	9.8e-05	SHOW WARNINGS
2	0.00184625	SELECT COUNT(*) FROM student
3	0.001534	SELECT COUNT(*) FROM student
4	0.001821	SELECT COUNT(*) FROM student
5	0.00120925	SELECT COUNT(*) FROM student
6	0.00125925	SELECT COUNT(*) FROM student
7	0.0016245	SELECT COUNT(*) FROM student
8	0.0011555	SELECT COUNT(*) FROM student

Запрос 2

```
SELECT COUNT(*)  
FROM student  
WHERE city_id =113;
```

Query_ID	Duration	Query
1	0.0001085	SHOW WARNINGS
2	0.00219425	SELECT COUNT(*) FROM student
3	0.002062	SELECT COUNT(*) FROM student WHERE city_id =113
4	0.00239775	SELECT COUNT(*) FROM student WHERE city_id =113
5	0.00186375	SELECT COUNT(*) FROM student WHERE city_id =113
6	0.002193	SELECT COUNT(*) FROM student WHERE city_id =113
7	0.002071	SELECT COUNT(*) FROM student WHERE city_id =113
8	0.002183	SELECT COUNT(*) FROM student WHERE city_id =113

Измерение времени выполнения запроса

Что можем посчитать?

Относительную разницу (%):

$$\frac{max_{\text{значение}} - min_{\text{значение}}}{max_{\text{значение}}} \cdot 100\%$$

Отношение (единицы):

$$\frac{max_{\text{значение}}}{min_{\text{значение}}}$$

Измерение времени выполнения запроса

Запрос 1

```
SELECT COUNT(*)  
FROM student;
```

Query_ID	Duration	Query
1	9.8e-05	SHOW WARNINGS
2	0.00184625	SELECT COUNT(*) FROM student
3	0.001534	SELECT COUNT(*) FROM student
4	0.001821	SELECT COUNT(*) FROM student
5	0.00120925	SELECT COUNT(*) FROM student
6	0.00125925	SELECT COUNT(*) FROM student
7	0.0016245	SELECT COUNT(*) FROM student
8	0.0011555	SELECT COUNT(*) FROM student

Средне время:

запрос 1: 0.001493

запрос 2: 0.002138

Оценка: относительная разница 30%, быстрее в 1.4 раза

Запрос 2

```
SELECT COUNT(*)  
FROM student  
WHERE city_id =113;
```

Query_ID	Duration	Query
1	0.0001085	SHOW WARNINGS
2	0.00219425	SELECT COUNT(*) FROM student
3	0.002062	SELECT COUNT(*) FROM student WHERE city_id =113
4	0.00239775	SELECT COUNT(*) FROM student WHERE city_id =113
5	0.00186375	SELECT COUNT(*) FROM student WHERE city_id =113
6	0.002193	SELECT COUNT(*) FROM student WHERE city_id =113
7	0.002071	SELECT COUNT(*) FROM student WHERE city_id =113
8	0.002183	SELECT COUNT(*) FROM student WHERE city_id =113

Измерение времени выполнения запроса

SHOW PROFILE

- показывает общую информацию
последнем запросе

Измерение времени выполнения запроса

SHOW PROFILE

- показывает общую информацию
последнем запросе

```
SELECT COUNT(*)  
FROM student;
```

Status	Duration
starting	0.000032
Executing hook on transaction	0.000010
starting	0.000023
checking permissions	0.000027
Opening tables	0.000031
init	0.000010
System lock	0.000011
optimizing	0.000010
statistics	0.000014
preparing	0.000015
executing	0.001138
end	0.000012
query end	0.000009
waiting for handler commit	0.000012
closing tables	0.000012
freeing items	0.000038
cleaning up	0.000014

```
SELECT COUNT(*)  
FROM student  
WHERE city_id =113;
```

Status	Duration
starting	0.000035
Executing hook on transaction	0.000011
starting	0.000011
checking permissions	0.000010
Opening tables	0.000025
init	0.000011
System lock	0.000012
optimizing	0.000013
statistics	0.000030
preparing	0.000021
executing	0.001812
end	0.000012
query end	0.000009
waiting for handler commit	0.000012
closing tables	0.000012
freeing items	0.000033
cleaning up	0.000014

Измерение времени выполнения запроса

SHOW PROFILE

[FOR QUERY *n*]

- показывает информацию о запросе с номером *n*

Измерение времени выполнения запроса

SHOW PROFILE [*type* [, *type*] ...] - Показывает информацию в соответствии
[**FOR QUERY** *n*] с параметрами профилирования ***type***

Измерение времени выполнения запроса

```
SHOW PROFILE [type [, type] ... ]  
[FOR QUERY n]
```

type: { **ALL** | полная информация
 BLOCK IO | количество операций ввода и вывода
 CONTEXT SWITCHES | количество переключений контекста
 CPU | время использования ЦП пользователем и системой
 IPC | количество отправленных и полученных сообщений
 PAGE FAULTS | количество ошибок страницы
 SOURCE | отображает имена функций
 SWAPS | количество свопов
}

Измерение времени выполнения запроса

SHOW PROFILE CPU;

Status	Duration	CPU_user	CPU_system
starting	0.000053	0.000033	0.000019
Executing hook on transaction	0.000010	0.000004	0.000007
starting	0.000013	0.000006	0.000006
checking permissions	0.000011	0.000011	0.000000
Opening tables	0.000027	0.000015	0.000013
init	0.000011	0.000004	0.000006
System lock	0.000013	0.000013	0.000000
optimizing	0.000011	0.000005	0.000007
statistics	0.000020	0.000013	0.000006
preparing	0.000021	0.000008	0.000013
executing	0.005826	0.003876	0.002481
end	0.000011	0.000011	0.000000
query end	0.000009	0.000002	0.000006
waiting for handler commit	0.000012	0.000012	0.000000
closing tables	0.000012	0.000006	0.000006
freeing items	0.000038	0.000011	0.000013
cleaning up	0.000015	0.000008	0.000006

Инструменты для анализа запросов

Для анализа запросов SQL необходимы инструменты, которые позволят оценить эффективность того или иного запроса, а затем его оптимизировать:

- измерение времени выполнения запроса;
- **оценка плана выполнения запроса.**

План выполнения запросов

План выполнения запроса - это последовательность шагов или инструкций СУБД, необходимых для выполнения SQL-запроса.

На каждом шаге операция, инициировавшая данный шаг выполнения SQL-запроса, извлекает строки данных, которые могут формировать конечный результат или использоваться для дальнейшей обработки.

Инструкции плана выполнения SQL-запроса представляются в виде последовательности операций, которые выполняются СУБД для запросов SQL SELECT, INSERT, DELETE, UPDATE.

План выполнения запросов

С помощью оператора:

```
EXPLAIN SQL_запрос;
```

Можно посмотреть план выполнения запроса интерпретатором SQL.

План выполнения запросов, информация

EXPLAIN SQL_запрос;

1. **id** – порядковый номер для каждого SELECT внутри запроса (когда имеется несколько подзапросов)
2. **select_type** – тип запроса SELECT.
 - **SIMPLE** — простой запрос SELECT без подзапросов или UNION
 - **PRIMARY** – данный SELECT – самый внешний запрос в JOIN
 - **DERIVED** – данный SELECT является частью подзапроса внутри FROM
 - **SUBQUERY** – первый SELECT в подзапросе
 - **DEPENDENT SUBQUERY** – подзапрос, который зависит от внешнего запроса
 - **UNCACHABLE SUBQUERY** – не кешируемый подзапрос
 - **UNION** – второй или последующий SELECT в UNION
 - **DEPENDENT UNION** – второй или последующий SELECT в UNION, зависимый от внешнего запроса
 - **UNION RESULT** – результат UNION

План выполнения запросов, информация

EXPLAIN SQL_запрос;

3. **table** – таблица, к которой относится выводимая строка.
4. **type** — указывает на то, как MySQL связывает используемые таблицы.
 - **System** – таблица имеет только одну строку;
 - **Const** – таблица имеет только одну соответствующую строку, которая проиндексирована. Это наиболее быстрый тип соединения потому, что таблица читается только один раз и значение строки может восприниматься при дальнейших соединениях как константа.
 - **Eq_ref** – все части индекса используются для связывания. Используемые индексы: PRIMARY KEY или UNIQUE NOT NULL. Это еще один наилучший возможный тип связывания.
 - **Ref** – все соответствующие строки индексного столбца считываются для каждой комбинации строк из предыдущей таблицы. Этот тип соединения для индексированных столбцов выглядит как использование операторов = или !=.
 - **Fulltext** – соединение использует полнотекстовый индекс таблицы.

План выполнения запросов, информация

EXPLAIN SQL_запрос;

3. **table** – таблица, к которой относится выводимая строка.
4. **type** — указывает на то, как MySQL связывает используемые таблицы.
 - **Ref_or_null** – то же самое, что и **ref**, но также содержит строки со значением **null** для столбца.
 - **Index_merge** – соединение использует список индексов для получения результирующего набора. Столбец **key** вывода команды **EXPLAIN** будет содержать список использованных индексов.
 - **Unique_subquery** – подзапрос **IN** возвращает только один результат из таблицы и использует первичный ключ.
 - **Index_subquery** – тоже, что и предыдущий, но возвращает более одного результата.
 - **Range** – индекс, использованный для нахождения соответствующей строки в определенном диапазоне, обычно, когда ключевой столбец сравнивается с константой, используя операторы вроде: **BETWEEN**, **IN**, **>**, **>=**.

План выполнения запросов, информация

EXPLAIN SQL_запрос;

3. **table** – таблица, к которой относится выводимая строка.
4. **type** — указывает на то, как MySQL связывает используемые таблицы.
 - **Index** – сканируется все дерево индексов для нахождения соответствующих строк.
 - **NULL** - Означает, что оптимизатор смог сам найти нужные данные для запроса и обращаться к таблице не требуется. *Например, для выборки минимального значения проиндексированного, в порядке возрастания столбца достаточно выбрать первое значение из индекса.*
 - **All** – Для нахождения соответствующих строк используются сканирование всей таблицы. **Это наихудший тип соединения и обычно указывает на отсутствие подходящих индексов в таблице.**

План выполнения запросов, информация

EXPLAIN SQL_запрос;

5. **Possible_keys** – показывает индексы, которые могут быть использованы для нахождения строк в таблице. На практике они могут использоваться, а могут и не использоваться. **Значение NULL указывает на то, что не найдено ни одного подходящего индекса** .

6. **Key**– указывает на использованный индекс. Этот столбец может содержать индекс, не указанный в столбце **possible_keys**. В процессе соединения таблиц оптимизатор ищет наилучшие варианты и может найти ключи, которые не отображены в **possible_keys**, но являются более оптимальными для использования.

7. **Key_len** – длина индекса, которую оптимизатор MySQL выбрал для использования. Например, если у вас есть **PRIMARY KEY ID** типа **INT**, то, при его использовании, **key_len** будет равен 4, потому что длина **INT** всегда равна 4 байта. В случае составных ключей **key_len** будет равен сумме байтов их типов.

План выполнения запросов, информация

EXPLAIN SQL_запрос;

8. **Ref** - показывает, какие столбцы или константы сравниваются с указанным в **key** индексом. Принимает значения **NULL**, **const** или название столбца другой таблицы.

9. **Rows** – показывает предполагаемое количество строк, которое, по мнению MySQL, будет прочитано. *Это число является приблизительным и может оказаться очень неточным. Оно вычисляется при каждой итерации плана выполнения с вложенными циклами. Часто это значение путают с количеством строк в результирующем наборе, что неверно, потому что столбец **rows** показывает количество строк, которые нужно будет просмотреть. При вычислении значения не учитываются буферы соединения, поэтому реальное число может быть намного меньше предполагаемого.*

10. **Filtered** - Показывает, какую долю от общего количества числа просмотренных строк вернет интерпретатор MySQL. *Максимальное значение 100, то есть будет возвращено все 100 % просмотренных строк. Если умножить эту долю на значение в столбце **rows**, то получится приблизительная оценка количества строк, которые MySQL будет соединять с последующими таблицами. Например, если в строке **rows** 100 записей, а значение **filtered** 50,00 (50 %, 0.5), то это число будет вычислено как $100 * 0.5 = 50$.*

План выполнения запросов, информация

EXPLAIN SQL_запрос;

11. **Extra** – содержит дополнительную информацию, относящуюся к плану выполнения запроса. Вот некоторые из них

- **Using filesort (!!!)**. Сервер MySQL вынужден прибегнуть к внешней сортировке, вместо той, что задаётся индексом. Сортировка может быть произведена как в памяти, так и на диске, о чем EXPLAIN никак не сообщает.
- **Using index (!!!)**. MySQL использует покрывающий индекс, чтобы избежать доступа к таблице
- **Using index condition (!!!)**. Информация считывается из индекса, чтобы затем можно было определить, следует ли читать строку целиком. Иногда стоит поменять местами условия в **WHERE**, чтобы **Using index condition** превратилось в **Using index**.
- **Using index for group-by (!!!)**. Похож на Using index, но для группировки GROUP BY или DISTINCT. Обращения к таблице не требуется, все данные есть в индексе.
- **Using temporary (!!!)**. Будет создана временная таблица для сортировки или группировки результатов запроса.
- **Using where (!!!)**. Сервер будет вынужден дополнительно фильтровать те строки, которые уже были отфильтрованы подсистемой хранения. Если вы встретили **Using where**, то стоит переписать запрос, используя другие возможные индексы.

План выполнения запросов, информация

EXPLAIN SQL_запрос;

Подробнее можно посмотреть здесь:

<https://habr.com/ru/company/citymobil/blog/545004/>

Оценка используемых ресурсов

Эксперимент

Проверим следующие гипотезы:

1. Наличие индекса увеличивает скорость выполнения запроса по одной таблице.
2. Уникальный индекс эффективнее полнотекстового.
3. Индекс неэффективен для небольших таблиц.

Оценка используемых ресурсов

Эксперимент

Алгоритм эксперимента:

1. Создадим блок запросов, который будет включать один и тот же запрос по таблице с индексом и без него:

```
SET profiling = 1;
```

```
# запрос
```

```
CREATE INDEX ...
```

```
# запрос
```

```
SHOW PROFILES;
```

2. Запустим этот блок 7 раз.
3. Оценим результаты, вычислив среднее время выполнения запросов.

Оценка используемых ресурсов

Эксперимент

Эмпирические данные:

Таблицы:

book (29 записей)

author (10 записей)

student (10145 запись)

city (1000 записей)

ВАЖНО! Для более надежного эксперимента нужна таблица с 1000000 записей.

Оценка используемых ресурсов

Эксперимент

Запросы:

```
SET profiling=1;
```

```
SELECT student_name
```

```
FROM student
```

```
WHERE student_name = "Студент_6312";
```

```
CREATE [UNIQUE] INDEX ind_student ON student (student_name);
```

```
SELECT student_name
```

```
FROM student
```

```
WHERE student_name = "Студент_6312";
```

```
SHOW PROFILES;
```

Оценка используемых ресурсов

Эксперимент (таблицы с маленьким количеством записей)

Запросы:

```
SET profiling=1;
```

```
SELECT title  
FROM book  
WHERE title = "Поэмы";
```

```
CREATE INDEX ind_title ON book (title);
```

```
SELECT title  
FROM book  
WHERE title = "Поэмы";
```

```
SHOW PROFILES;
```

Оценка используемых ресурсов

Эксперимент, гипотеза 1 (индекс ускоряет работу запроса)

гипотеза 2 (уникальные индекс эффективнее)

Запрос	Средняя длительность	Отношение
Запрос без индекса (1145 записей)	0,00634455	1
Запрос с индексом (1145 записей)	0,00048515	в 13 раз
Запрос с уникальным индексом (1145 записей)	0,00038392	в 16,5 раз

Вывод: гипотеза подтверждаются (но лучше проверить на больших данных)

Оценка используемых ресурсов

Эксперимент, гипотеза 1 (индекс ускоряет работу запроса)

гипотеза 2 (уникальные индекс эффективнее)

План выполнения запросов:

Характеристика	Запрос без индекса	Запрос с индексом	Запрос с уникальным индексом
скорость	1	в 13 раз быстрее	в 16.5 раз быстрее
selected_type	SIMPLE	SIMPLE	SIMPLE
table	student	student	student
partitions	NULL	NULL	NULL
type	ALL	ref	const
posible_keys	NULL	ind_student	ind_student
key	NULL	ind_student	ind_student
key_len	NULL	161	161
ref	NULL	const	const
rows	10145	1	1
filtered	10.0	100.0	100.0
Extra	Using where	Using where; Using index	Using index

Оценка используемых ресурсов

Эксперимент, гипотеза 1 (индекс ускоряет работу запроса)

гипотеза 2 (уникальные индекс эффективнее)

Плюс

Это означает полное сканирование таблицы, то есть MySQL будет просматривать строку за строкой...

Характеристика	Запрос без индекса	Запрос с индексом	Запрос с уникальным индексом
скорость	1	в 13 раз быстрее	в 16.5 раз быстрее
selected_type	SIMPLE	SIMPLE	SIMPLE
table	student	student	student
partitions	NULL	NULL	NULL
type	ALL	ref	const
posible_keys	NULL	ind_student	ind_student
key	NULL	ind_student	ind_student
key_len	NULL	161	161
ref	NULL	const	const
rows	10145	1	1
filtered	10.0	100.0	100.0
Extra	Using where	Using where; Using index	Using index

Оценка используемых ресурсов

Эксперимент, гипотеза 1 (индекс ускоряет работу запроса)

гипотеза 2 (уникальный индекс эффективнее)

План выполнения 30

Поиск по индексу, в результате которого возвращаются все строки, соответствующие единственному заданному значению. Применяется в случаях, если ключ не является уникальным

Характеристика	Запрос 6	Запрос 7	Запрос с уникальным индексом
скорость			в 16.5 раз быстрее
selected_type	SIMPLE	SIMPLE	SIMPLE
table	student	student	student
partitions	NULL	NULL	NULL
type	ALL	ref	const
posible_keys	NULL	ind_student	ind_student
key	NULL	ind_student	ind_student
key_len	NULL	161	161
ref	NULL	const	const
rows	10145	1	1
filtered	10.0	100.0	100.0
Extra	Using where	Using where; Using index	Using index

Оценка используемых ресурсов

Эксперимент, гипотеза 1 (индекс ускоряет работу запроса)

гипотеза 2 (уникальные индекс эффективнее)

План выполнения запросов:

Таблица содержит не более одной совпадающей строки. Это значит, что оптимизатору MySQL удалось привести запрос к константе

Характеристика	Запрос без индекса	Запрос с индексом	Запрос с уникальным индексом
скорость	1	в 13 раз быстрее	в 16.5 раз быстрее
selected_type	SIMPLE	SIMPLE	SIMPLE
table	student	student	student
partitions	NULL	NULL	NULL
type	ALL	ref	const
posible_keys	NULL	ind_student	ind_student
key	NULL	ind_student	ind_student
key_len	NULL	161	161
ref	NULL	const	const
rows	10145	1	1
filtered	10.0	100.0	100.0
Extra	Using where	Using where; Using index	Using index

Оценка используемых ресурсов

Эксперимент, гипотеза 1 (индекс ускоряет работу запроса)

гипотеза 2 (уникальные индекс эффективнее)

План выполнения запросов:

Характеристика	Запрос без индекса	Запрос с индексом	Запрос с уникальным индексом
скорость	1	в 13 раз быстрее	в 16.5 раз быстрее
selected_type	SIMPLE	SIMPLE	SIMPLE
table	student	student	student
partitions	NULL	NULL	NULL
type	ALL	ref	const
posible_keys	NULL	ind_student	ind_student
key	NULL	ind_student	ind_student
key_len	NULL	161	161
ref	NULL	const	const
rows	10145	1	1
filtered	10.0	100.0	100.0
Extra	Using where	Using where; Using index	Using index

Оценка используемых ресурсов

Эксперимент, гипотеза 1 (индекс ускоряет работу запроса)

гипотеза 2 (уникальные индекс эффективнее)

План выполнения запросов:

Характеристика	Запрос без индекса	Запрос с индексом	Запрос с уникальным индексом
скорость	1	в 13 раз быстрее	в 16.5 раз быстрее
selected_type	SIMPLE	SIMPLE	SIMPLE
table	student	student	student
partitions	Приблизительное количество просматриваемых строк $10145 * 0.1 = 1015$		NULL
type			const
posible_keys		ind_student	ind_student
key	NULL	ind_student	ind_student
key_len	NULL	161	161
ref	NULL	const	const
rows	10145	1	1
filtered	10.0	100.0	100.0
Extra	Using where	Using where; Using index	Using index

Оценка используемых ресурсов

Эксперимент, гипотеза 3 (на маленьких таблицах индекс не эффективен)

Запрос	Средняя длительность	Отношение
Запрос без индекса (29 записей)	0,00032215	в 1,3 раза (24%)
Запрос с индексом (29 записей)	0,000424	1

Вывод: гипотеза подтверждаются (но лучше проверить на больших данных)

Оценка используемых ресурсов

Эксперимент, гипотеза 3 (индекс ускоряет работу запроса)
гипотеза 2 (уникальные индекс эффективнее)

План выполнения запросов (ищем «Поэмы», 2 записи):

Характеристика	Запрос без индекса	Запрос с индексом
скорость	в 1,3 раз быстрее	1
selected_type	SIMPLE	SIMPLE
table	book	book
partitions	NULL	NULL
type	ALL	ref
posible_keys	NULL	ind_title
key	NULL	ind_title
key_len	NULL	323
ref	NULL	CONST
rows	1	2
filtered	100.0	100.0
Extra	Using where	Using index

Оценка используемых ресурсов

Эксперимент, гипотеза 3 (индекс ускоряет работу запроса)

гипотеза 2 (уникальные индекс эффективнее)

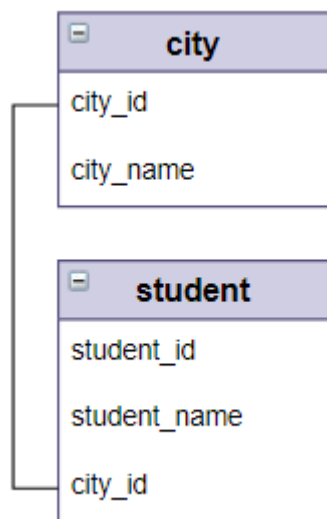
План выполнения запросов (ищем «Двенадцать стульев», 1 запись):

Характеристика	Запрос без индекса	Запрос с индексом
скорость	в 1,15 раз быстрее	1
selected_type	SIMPLE	SIMPLE
table	book	book
partitions	NULL	NULL
type	ALL	ref
posible_keys	NULL	ind_title
key	NULL	ind_title
key_len	NULL	323
ref	NULL	CONST
rows	29	1
filtered	10.0	100.0
Extra	Using where	Using index

Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Логическая схема базы данных

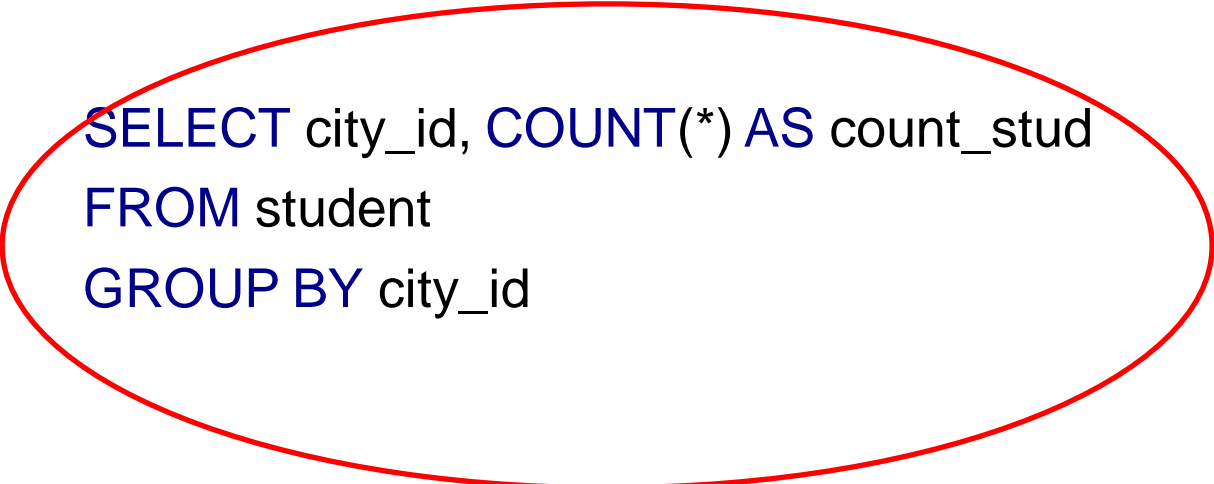


Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Вариант 1 (два вложенных запроса)

Сколько студентов
проживает в каждом
городе



```
SELECT city_id, COUNT(*) AS count_stud  
FROM student  
GROUP BY city_id
```

Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Вариант 1 (два вложенных запроса)

```
SELECT MAX(count_stud)
FROM (
    SELECT city_id, COUNT(*) AS count_stud
    FROM student
    GROUP BY city_id
) query_in
```

Находит максимальное
из полученных
значений

Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Вариант 1 (два вложенных запроса)

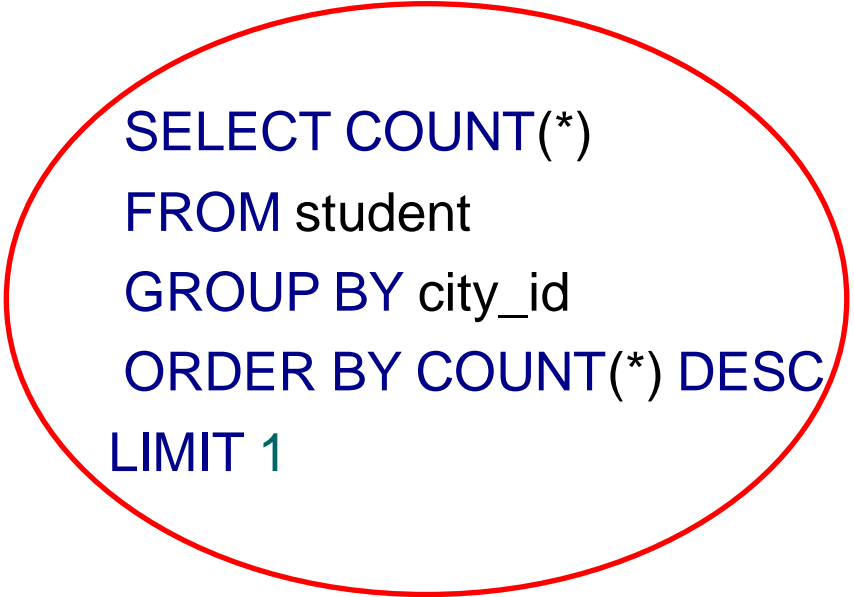
```
SELECT city_name, COUNT(*)  
FROM  
    city JOIN student USING(city_id)  
GROUP BY city_name  
HAVING COUNT(*) = (  
    SELECT MAX(count_stud)  
    FROM (  
        SELECT city_id, COUNT(*) AS count_stud  
        FROM student  
        GROUP BY city_id  
    ) query_in  
);
```

Выводит города, в
которых количество
проживающих равно
максимальному

Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Вариант 2 (один вложенный запрос)



```
SELECT COUNT(*)  
FROM student  
GROUP BY city_id  
ORDER BY COUNT(*) DESC  
LIMIT 1
```

- Вычисляет количество студентов, проживающих в каждом городе
- Сортирует по убыванию количества
- Ограничивает выборку одной записью

Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Вариант 2 (один вложенный запрос)

```
SELECT city_name, COUNT(*)  
FROM  
    city JOIN student USING(city_id)  
GROUP BY city_name  
HAVING COUNT(*) = (  
    SELECT COUNT(*)  
    FROM student  
    GROUP BY city_id  
    ORDER BY COUNT(*) DESC  
    LIMIT 1  
);
```

Выводит города, в
которых количество
проживающих равно
максимальному

Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Вариант 3 (два табличных выражений)

```
WITH get_count(city_id, count_id)
AS( SELECT city_id, COUNT(*)
      FROM student
      GROUP BY city_id
    ),
get_max_count(count_max)
AS(
  SELECT MAX(count_id) FROM get_count
)
SELECT city_name, COUNT(*)
FROM get_max_count, city JOIN student USING(city_id)
GROUP BY city_name, count_max
HAVING COUNT(*) = count_max;
```


Оценка используемых ресурсов

Пример. Вывести город, в котором проживает максимальное количество студентов.

Вариант 4 (одно табличное выражение)

```
WITH get_max_count(count_max)
AS(
    SELECT COUNT(*)
    FROM student
    GROUP BY city_id
    ORDER BY COUNT(*)
    DESC LIMIT 1
)
SELECT city_name, COUNT(*)
FROM get_max_count,
    city JOIN student USING (city_id)
GROUP BY city_name, count_max
HAVING COUNT(*) = count_max;
```

Оценка используемых ресурсов

Эксперимент

1. Создадим блок запросов, который будет включать все 4 варианта реализации:

```
SET profiling = 1;
```

```
# вариант 1
```

```
# вариант 2
```

```
# вариант 3
```

```
# вариант 4
```

```
SHOW PROFILES;
```

2. Запустим этот блок 7 раз.
3. Оценим результаты, вычислив среднее время выполнения запросов.

Оценка используемых ресурсов

Эксперимент, оценка результата

Запрос	Средняя длительность	Разница
Вариант 1 (два вложенных запроса)	0,01978735	1
Вариант 2 (один вложенный запрос)	0,01899985	4%
Вариант 3 (два табличных выражения)	0,0192881	3%
Вариант 4 (одно табличное выражение)	0,0189771	4%

Оценка используемых ресурсов

Эксперимент, оценка результата

Запрос	Средняя длительность	Разница
Вариант 1 (два вложенных запроса)	0,01978735	
Вариант 2 (один вложенный запрос)	0,01899985	4%
Вариант 3 (два табличных выражения)	0,0192881	3%
Вариант 4 (одно табличное выражение)	0,0189771	4%

Результаты не позволяют сделать вывод о том, какой запрос более эффективен.

Но и нельзя сказать, что они одинаковы по эффективности. Необходимо изменить условия эксперимента, например, протестировать эти запросы на базе, в которой несколько миллионов записей.

Оценка используемых ресурсов

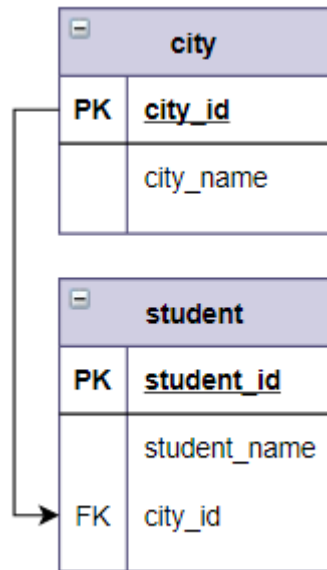
Эксперимент, оценка результата

	id	selected_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
Запрос 1 (два вложенных)	1	PRIMARY	city	NULL	ALL	NULL	NULL	NULL	NULL	1000	100	Using temporary
	1	PRIMARY	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	10	Using where; Using join buffer (hash join)
	2	SUBQUERY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	NULL
	3	DERIVED	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	Using temporary
Запрос 2 (один вложенный)	1	PRIMARY	city	NULL	ALL	NULL	NULL	NULL	NULL	1000	100	Using temporary
	1	PRIMARY	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	10	Using where; Using join buffer (hash join)
	2	SUBQUERY	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	Using temporary; Using filesort

Оценка используемых ресурсов

Эксперимент 2.

1. Изменим структуру таблиц, добавив в них первичные и внешние ключи:



2. Проведем описанный выше эксперимент с новой структурой.
3. Сравним результаты экспериментов.

Оценка используемых ресурсов

Эксперимент 2.

Запрос	Средняя длительность		Разница
	Без ключей	С ключами	
Вариант 1 (два вложенных запроса)	0,01978735	0,0160982	19%
Вариант 2 (один вложенный запрос)	0,01899985	0,01545595	19%
Вариант 3 (два табличных выражения)	0,0192881	0,01637125	15%
Вариант 4 (одно табличное выражение)	0,0189771	0,0156346	18%

Результаты позволяют сделать **ОСТОРОЖНЫЙ** (так как данных все таки мало) вывод о том, что запросы с ключами более эффективные (средняя разница составляет 18%, запрос с ключами в среднем работает в 1.2 раза быстрее).

Оценка используемых ресурсов

Эксперимент, оценка результата

	id	selected_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
Запрос 1 (два вложенных)	1	PRIMARY	city	NULL	ALL	NULL	NULL	NULL	NULL	1000	100	Using temporary
	1	PRIMARY	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	10	Using where; Using join buffer (hash join)
	2	SUBQUERY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	NULL
	3	DERIVED	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	Using temporary
Запрос 1 (два вложенных, PRIMARY KEY)	1	PRIMARY	city	NULL	ALL	PRIMARY	NULL	NULL	NULL	2	50	Using where; Using join buffer (hash join)
	1	PRIMARY	student	NULL	index	city_id	city_id	5	NULL	10145	100	Using index; Using temporary
	2	SUBQUERY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	NULL
	3	DERIVED	student	NULL	index	city_id	city_id	5	NULL	10145	100	Using index

Методы оптимизации

Основные методы оптимизации:

1. Индексация в базах данных.
2. **Изменение структуры базы данных.**

Изменение структуры базы данных

Если индексы не помогают, запрос все равно работает медленно (много вложенных запросов, сортировки и пр.), можно добавить новые таблицы или столбцы в уже существующие таблицы, чтобы упростить запрос.

Изменение структуры базы данных

Например, в примере с поиском города, в котором проживает больше всего студентов, что делают запросы:

- каждый раз выполняют группировку, чтобы найти количество студентов, проживающих в каждом городе;
- каждый раз осуществляется поиск максимума.

Если бы в таблицу **city** было бы всегда актуальное количество, проживающих в городе студентов, то объявив упорядоченный по убыванию индекс по количеству проживающих, поиск максимума осуществлялся бы мгновенно (всего то нужно взять первую строку таблицы...).

Изменение структуры базы данных

Каким образом можно получить столбец с актуальным количеством?

- если добавляется новый студент в каком-то городе – увеличить количество студентов на 1 (нужно найти и обновить всего одну запись!);
- если студент уезжает из города – уменьшить количество студентов в этом городе на 1 (обновляется 1 запись);
- если студент переезжает из одного города в другой – в одном городе количество уменьшить на 1, в другом – увеличить на 1 (обновляется 2 записи);
- делать это автоматически при выполнении запросов на ДОБАВЛЕНИЕ, УДАЛЕНИИ и ОБНОВЛЕНИИ записей.

Триггеры SQL

Триггер позволяет описать действия, которые должны быть выполнены при наступлении определенных действий с данными в таблице.

Триггеры по своей сути представляют обработчики событий, которые выполняются, когда осуществляется удаление, вставка или обновление данных в таблице.

Например, при добавлении новой записи в одну таблицу, можно изменить данные в других таблицах.

Триггеры SQL

Перед созданием триггера необходимо ответить на следующие вопросы:

1. С какой таблицей он будет связан?
2. В какой момент времени он должен быть "запущен" и с какой операцией ассоциируется?
3. Что именно он должен делать?

Оптимизация за счет изменения структуры

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Алгоритм

1. Добавить в таблицу **city** столбец **city_number**.
2. Создать упорядоченный индекс по столбцу **city_number**.
3. Занести в **city_number** текущее количество студентов в каждом городе.
4. Создать триггеры, которые при добавлении, удалении или переезде студентов обновляют столбец **city_number**.

Оптимизация за счет изменения структуры

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 1. Добавить в таблицу **city** столбец **city_number**.

```
ALTER TABLE city ADD city_number INT;
```

Шаг 2. Создать упорядоченный индекс по столбцу **city_number**.

```
CREATE INDEX ind_number ON city(city_number DESC);
```


Оптимизация за счет изменения структуры

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 3. Занести в city_number текущее количество студентов в каждом городе.

```
WITH get_count(city_id, count_student)
AS(
    SELECT city_id, COUNT(*)
    FROM student
    GROUP BY city_id
)
UPDATE
    city JOIN get_count USING(city_id)
SET city_number = count_student;
```

Оптимизация за счет изменения структуры

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 4. Создать триггер, который при добавлении студентов обновляет столбец **city_number**.

```
CREATE TRIGGER inc_numder_city_insert  
AFTER INSERT  
ON student  
FOR EACH ROW  
BEGIN  
    UPDATE city  
    SET city_number = city_number + 1  
    WHERE city_id = NEW.city_id;  
END;
```

Оптимизация за счет изменения структуры

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 4. Создать триггер, который при удалении студента обновляет столбец **city_number**.

```
CREATE TRIGGER dec_numder_city_delete  
AFTER DELETE  
ON student  
FOR EACH ROW  
BEGIN  
    UPDATE city  
    SET city_number = city_number - 1  
    WHERE city_id = OLD.city_id;  
END;
```

Оптимизация за счет изменения структуры

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 4. Создать триггер, который при переезде студента обновляет столбец **city_number**.

```
CREATE TRIGGER inc_dec_numder_city
AFTER UPDATE
ON student
FOR EACH ROW
BEGIN
    UPDATE city
    SET city_number = city_number - 1
    WHERE city_id = OLD.city_id;
    UPDATE city SET city_number = city_number + 1
    WHERE city_id = NEW.city_id;
END;
```

Оптимизация за счет изменения структуры

Пример. Найти город с максимальной численностью студентов.

```
SELECT
    city_name, city_number
FROM city
WHERE city_number = (
    SELECT MAX(city_number)
    FROM city
);
```

Оценка используемых ресурсов

Пример. Найти город с максимальным количеством студентов.

Запрос	Средняя длительность	Изменение	
		1	
Запросы (с вложенными) без ключей	0,0192631	1	
Запросы (с вложенными) с ключами	0,01589	1,2	1
Запрос для таблицы измененной структуры	0,0007253	26,6	22

Оценка используемых ресурсов

оценка результата

	id	selected_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
Запрос (два вложенных)	1	PRIMARY	city	NULL	ALL	NULL	NULL	NULL	NULL	1000	100	Using temporary
	1	PRIMARY	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	10	Using where; Using join buffer (hash join)
	2	SUBQUERY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	NULL
	3	DERIVED	student	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	Using temporary
Запрос (два вложенных, PRIMARY KEY)	1	PRIMARY	city	NULL	ALL	PRIMARY	NULL	NULL	NULL	2	50	Using where; Using join buffer (hash join)
	1	PRIMARY	student	NULL	index	city_id	city_id	5	NULL	10145	100	Using index; Using temporary
	2	SUBQUERY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	10145	100	NULL
	3	DERIVED	student	NULL	index	city_id	city_id	5	NULL	10145	100	Using index
Запрос по измененной таблице	1	PRIMARY	city	NULL	ref	ind_number	ind_number	5	CONST	1	100	Using where
	2	SUBQUERY	city	NULL	index	NULL	ind_number	5	NULL	1000	100	Using index

Методы оптимизации

Основные методы оптимизации:

1. Индексация в базах данных.
2. Изменение структуры базы данных.
3. **Использование временных таблиц.**

Создание временных таблиц

Периодически, особенно при работе с результатами вычислений или сводными данными, проще и быстрее

- один раз запустить сложный запрос на выборку всех необходимых данных,
- сохранить его результаты во временную таблицу
- дальше работать уже с ней,

Это позволит не запускать каждый раз сложный и медленный запрос.

Создание временных таблиц

Временная таблица - это объект базы данных, который хранится и управляется системой базы данных на временной основе.

Временная таблица принадлежит создавшему ее сеансу, и доступна только этому сеансу. Временная таблица удаляется по завершению создавшего ее сеанса.

Это означает, что два разных сеанса могут использовать одно и то же имя временной таблицы без конфликта друг с другом или с существующей не временной таблицей с тем же именем.

Создание временных таблиц

Синтаксис создания временной таблицы:

```
CREATE TEMPORARY TABLE tmp_city  
SELECT ...
```

Для временной таблицы можно устанавливать ключи и индексы.

Оптимизация, временные таблицы

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Алгоритм

1. Создать временную таблицу, в которой посчитать количество студентов в каждом городе.
2. Добавить упорядоченный индекс по столбцу с численностью студентов.
3. Создать временную таблицу, в которой выполнить необходимые вычисления (найти минимум, максимум, среднее).
4. Реализовать запросы на выборку с отбором нужных городов.

Оптимизация, временные таблицы

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 1. Создать временную таблицу, в которой посчитать количество студентов в каждом городе.

```
CREATE TEMPORARY TABLE tmp_city  
SELECT city_id, city_name,  
       COUNT(*) AS city_number  
FROM  
    city  
    JOIN student USING(city_id)  
GROUP BY city_id;
```

Оптимизация, временные таблицы

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 2. Добавить упорядоченный индекс по столбцу с численностью студентов.

```
CREATE INDEX ind_city ON tmp_city (city_number DESC);
```

Оптимизация, временные таблицы

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 3. Создать временную таблицу, в которой выполнить необходимые вычисления (найти минимум, максимум, среднее).

```
CREATE TEMPORARY TABLE tmp_city_result  
SELECT  
    MIN(city_number) min_number,  
    MAX(city_number) max_number,  
    AVG(city_number) avg_number  
FROM tmp_city;
```

Оптимизация, временные таблицы

Пример. Необходимо актуализировать информацию о количестве студентов, проживающих в городе.

Шаг 4. Реализовать запросы на выборку с отбором нужных городов.

```
SELECT city_name, city_number  
FROM tmp_city, tmp_city_result  
WHERE city_number = max_number;
```

```
SELECT city_name, city_number  
FROM tmp_city, tmp_city_result  
WHERE city_number = min_number;
```

```
SELECT city_name, city_number  
FROM tmp_city, tmp_city_result  
WHERE city_number < avg_number/10;
```


Оценка используемых ресурсов

Пример. Найти город с максимальным количеством студентов.

Запрос	Средняя длительност ь	Изменение		
Запросы (с вложенными) без ключей	0,0192631	1		
Запросы (с вложенными) с ключами	0,01589	1,2	1	
Запрос для таблицы измененной структуры	0,0007253	26,6	22	1
Запрос с временными таблицами	0,00029075	66	54	2,5

Оценка используемых ресурсов

оценка результата

	id	selected_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
Запрос(поиск минимума или максимума)	1	SIMPLE	tmp_city_result	NULL	ALL	NULL	NULL	NULL	NULL	1	100	Using WHERE
	1	SIMPLE	tmp_city	NULL	ref	ind_city	ind_city	8	...max_number	1	100	NULL
Запрос (ниже среднего)	1	SIMPLE	tmp_city_result	NULL	ALL	NULL	NULL	NULL	NULL	1	100	NULL
	1	SIMPLE	tmp_city	NULL	ALL	NULL	NULL	NULL	NULL	870	33	Range checked for each record (index map: 0x1)

Методы оптимизации

Основные методы оптимизации:

1. Использование индексов
2. Изменение структуры базы данных
3. Использование временных таблиц
4. **Денормализация базы данных**

Денормализация базы данных

Денормализация базы данных – это стратегия, которую применяют к уже нормализованной базе данных с целью повышения ее производительности.

Смысл этого действия — поместить избыточные данные туда, где они смогут принести максимальную пользу.

Для этого можно использовать дополнительные поля в уже существующих таблицах, добавлять новые таблицы или даже создавать новые экземпляры существующих таблиц.

Логика в том, чтобы снизить время исполнения определенных запросов через упрощение доступа к данным или через создание таблиц с результатами отчетов, построенных на основании исходных данных.

Денормализация базы данных

Непременное условие процесса денормализации — наличие нормализованной базы.

Важно понимать различие между ситуацией, когда база данных вообще не была нормализована, и нормализованной базой, прошедшей затем денормализацию.

Во втором случае — все хорошо, а вот первый говорит об ошибках в проектировании или недостатке знаний у специалистов, которые этим занимались.

Денормализация базы данных

Простейший пример – это включение вычисляемого столбца в таблицу **city**.

В нормализованной базе данных вычисляемых столбцов быть не должно.

Денормализация базы данных

Пример с созданием нескольких таблиц.

Пусть таблица **Продажи** содержит большое количество данных, при этом происходит очень частое обращение (поиск, обновление, удаление и пр. записей).

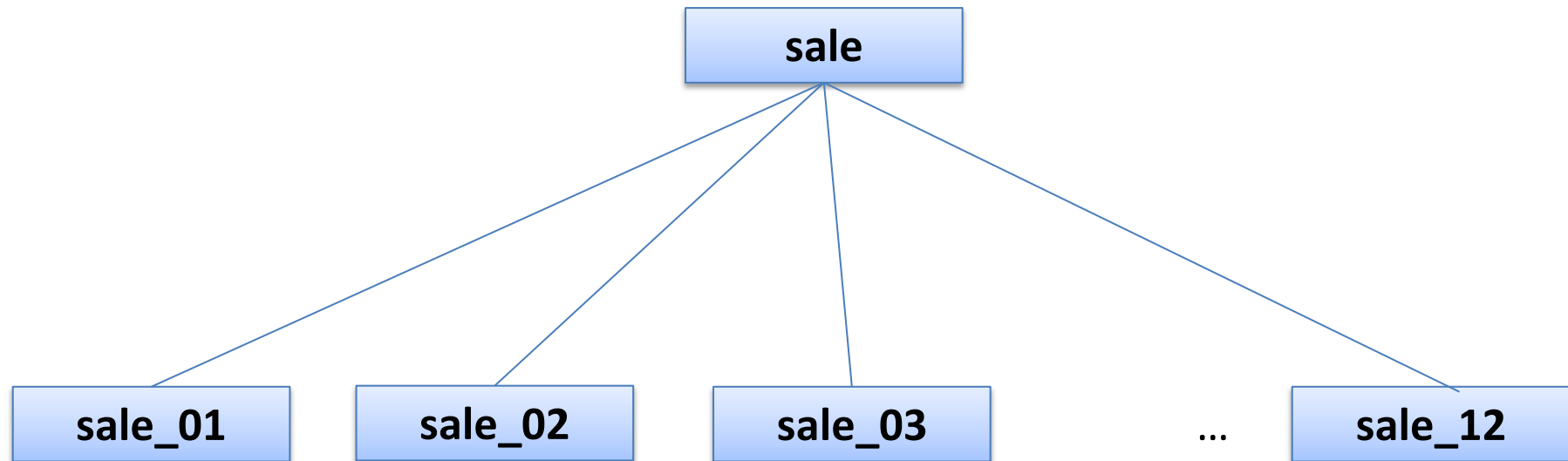
Включение индексов и другие методы не помогают.

Вариант решения: создать для каждого месяца отдельную таблицу.

И при обращении к базе сначала выбирать нужную таблицу (по номеру месяца), а затем уже в этой таблице выполнять нужные действия.

Денормализация базы данных

Вариант решения: создать для каждого месяца отдельную таблицу.



Для оптимизации запроса можно использовать следующие инструменты:

1. Оценка времени выполнения (`SET profiling=1; ...SHOW PROFILES`)
2. План выполнения запросов (`EXPLAIN`)

Основные методы оптимизации:

1. Создание индексов
2. Изменение структуры базы данных
3. Временные таблицы
4. Денормализация базы данных



Спасибо за внимание!