

## Лекция 11.

**Производительность запросов.  
Индексирование базы данных**

# Как работает SQL-запрос?

Запрос

```
select name, email from users limit 10;
```



Магия

name	email
Devonte Stamm	marianne.krajcik@bartoletti.com
Merritt Grimes	rempel.yvette@kertzmann.com
Dianna Herzog	jarrell.stokes@gmail.com
Jamel Nader	edgar.bayer@kohler.com
Immanuel Bode	jamal68@yahoo.com
Dwight Reichel	camron36@yahoo.com
Holden Lueilwitz	hdaugherty@hotmail.com
Pablo Kuhlman	arden.lemke@leffler.com
Roscoe Rice	katelynn.heathcote@thompson.org
Arlene Harvey	crist.shannon@hansen.com

Результат

**SQL - это язык программирования?**

# SQL - язык программирования?

Программа

```
select name, email from users limit 10;
```



Магия  
Компиляция/  
Интерпретация

name	email
Devonte Stamm	marianne.krajcik@bartoletti.com
Merritt Grimes	rempel.yvette@kertzmann.com
Dianna Herzog	jarrell.stokes@gmail.com
Jamel Nader	edgar.bayer@kohler.com
Immanuel Bode	jamal68@yahoo.com
Dwight Reichel	camron36@yahoo.com
Holden Lueilwitz	hdaugherty@hotmail.com
Pablo Kuhlman	arden.lemke@leffler.com
Roscoe Rice	katelynn.heathcote@thompson.org
Arlene Harvey	crist.shannon@hansen.com

Результат

Для описания ЯП нужно определить:

- **Лексику** – набор базовых символов и слов, из которых строится программа. Включает в себя:
  - **алфавит** - набор допустимых символов
  - **лексемы (токены)** - минимальные значимые единицы языка (ключевые слова, идентификаторы, операторы, разделители и т.д.)
- **Синтаксис** – правила построения корректных конструкций (допустимых текстов) из лексем. Формальный способ описания синтаксиса языка называется **грамматикой**.
- **Семантику** – правила, определяющие действия, которые выполнит компьютер под управлением программы.

- Лексика — это атомарные элементы языка (токены).
  - Синтаксис — правила построения из них корректных программ.
  - Грамматика — формализованное описание этих правил.
- 👉 Грамматика описывает синтаксис, а синтаксис применяется к лексемам.

# Анализ текста программы

Текст:

- **Предложения** (грамматика - синтаксис)
  - **Слова** (грамматика - лексика)
  - **Буквы** (алфавит)
- 
- **Лексический анализ** – разделение выражения на отдельные лексемы. Также анализатор может удалять из выражения комментарии, лишние разделители и т.д.
  - **Синтаксический анализ** – проверка правильности выражений, составленных из лексем.

## - Текстовые формы Бэкуса-Наура (БНФ)

### ✓ Преимущества БНФ/EBNF:

- Формальность и точность.
- Подходит для автоматической генерации парсеров.
- Широко используется в стандартах языков (C, Pascal, SQL, XML и др.).

## - Визуальные синтаксические диаграммы

### ✓ Преимущества диаграмм:

- Наглядность — легко понять даже новичку.
- Удобны для документации и обучения.
- Хорошо передают структуру сложных конструкций.

# **Описание грамматики языка программирования через БНФ**

# Формы Бэкуса-Наура (БНФ)

- **Терминальные символы** — это отдельные символы или их последовательности, имеющие конкретные известные значения и являющиеся неразрывным целым, не сводимым к другим символам.
- **Нетерминальные символы (метапеременные)** – элементы языка, не имеющие заранее известного значения (формулы, команды и т.п.), которые раскрываются правилам грамматики.
- ::= "есть по определению"
- | "или"
- {a} повтор a 0 или более раз
- [a] a входит 0 или 1 раз

# Формулы БНФ

**<идентификатор> ::= выражение**

- **идентификатор** – имя метапеременной
- **выражение** – комбинация терминальных символов и метапеременных.

**<Separator> ::= '.'** (разделитель для дробной части числа)

**<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'**

**<Unsigned> ::= <Digit> | <Digit><Unsigned>** (целое без знака)

**<Unsigned> ::= <Digit>{<Digit>}** (целое без знака)

## Примеры БНФ

<Assignment> ::= <Var> ':=' <Expression> (синтаксис оператора присваивания в Pascal)

<if> ::= 'if' <Condition> 'then' <Operator> ['else' <Operator>]

<for> ::= 'for' <Var> ':=' <Expression> ('to' | 'downto')  
<Expression> 'do' <Operator>

# Основы БНФ

**Задача.** Описать с помощью БНФ синтаксис вещественного числа.

- Перед числом может стоять знак — плюс или минус.
- Затем идет одна или несколько цифр.
- Потом может следовать точка, после которой будет еще одна или несколько цифр. Затем может быть указан показатель степени "E" (большое или малое), после которого может стоять знак плюс или минус, а затем должна быть одна или несколько цифр.

$$\langle \text{Number} \rangle ::= [\langle \text{Sign} \rangle] \langle \text{Digit} \rangle \{ \langle \text{Digit} \rangle \} [ \langle \text{Separator} \rangle \langle \text{Digit} \rangle \{ \langle \text{Digit} \rangle \} ] [ \langle \text{Exponent} \rangle [ \langle \text{Sign} \rangle] \langle \text{Digit} \rangle \{ \langle \text{Digit} \rangle \} ]$$
$$\langle \text{Digit} \rangle ::= '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9'$$
$$\langle \text{Sign} \rangle ::= '+' \mid '-'$$
$$\langle \text{Separator} \rangle ::= '.'$$
$$\langle \text{Exponent} \rangle ::= 'E' \mid 'e'$$

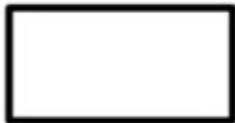
# **Описание грамматики ЯП с помощью синтаксических диаграмм**

# Синтаксические диаграммы

Графическое представление тех же формул и правил.  
Прямоугольники и овалы соединяются стрелками в нужной последовательности



Стрелка указывает направление движения.



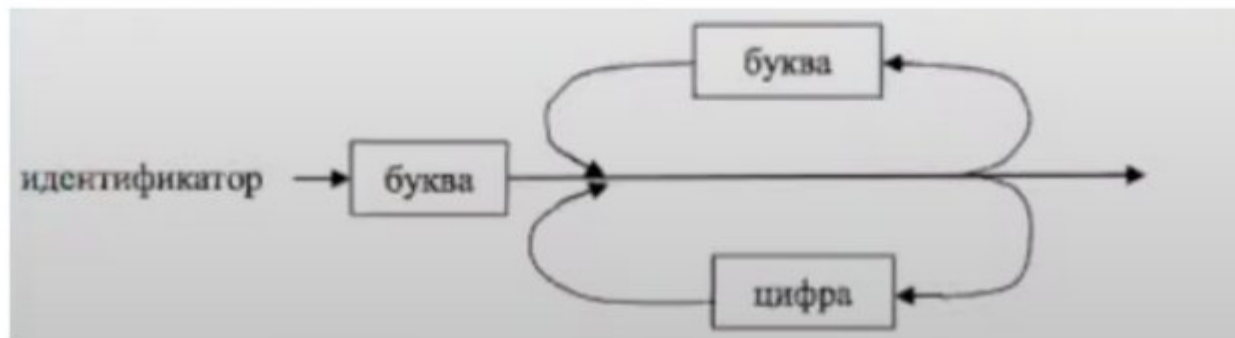
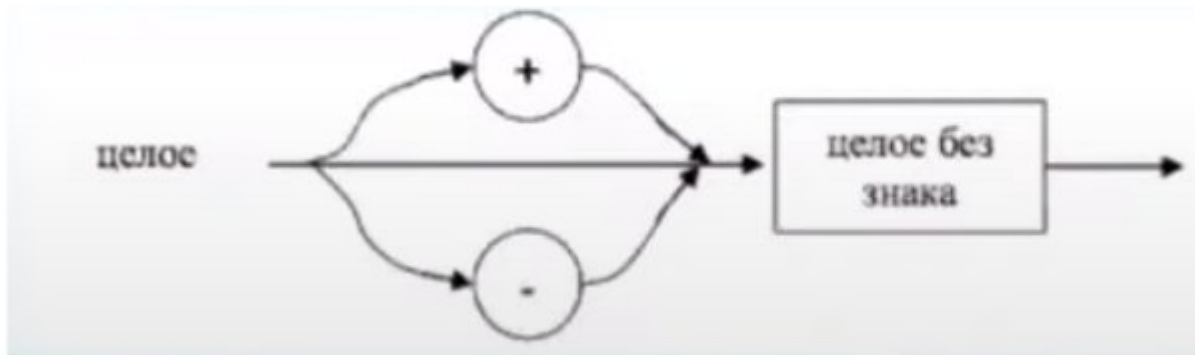
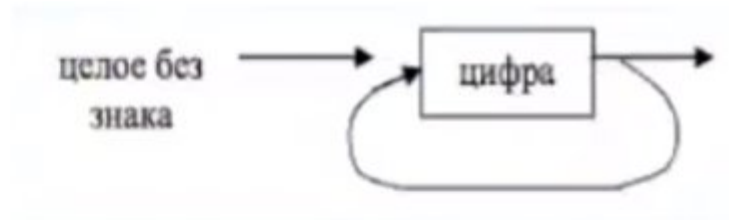
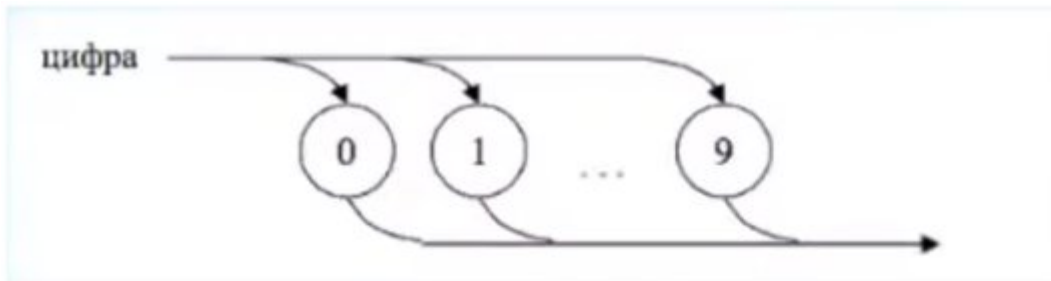
Прямоугольник соответствует нетерминальному символу.



Овал или круг соответствует терминальному символу.

# Синтаксические диаграммы

Текст удовлетворяет диаграмме, если, двигаясь по стрелкам от левого конца, мы найдем путь к правому концу.



# Описание грамматики SQL

# Описания правил грамматики SQL

- **Текстовые формы Бэкуса-Наура (БНФ)**

Стандарт SQL-2016

<https://jakewheat.github.io/sql-overview/sql-2016-foundation-grammar.html>

- **Визуальные синтаксические диаграммы**

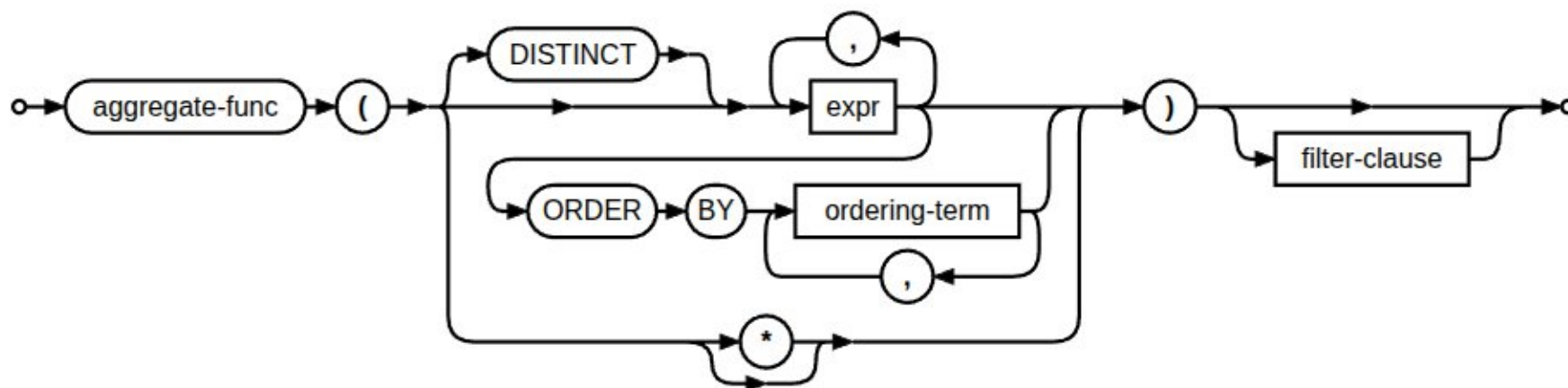
Реализация в SQLite

<https://www.sqlite.org/syntaxdiagrams.html>



## Syntax Diagrams For SQLite

aggregate-function-invocation:



References: [expr](#) [filter-clause](#) [ordering-term](#)

See also: [lang\\_aggfunc.html](#) [lang\\_expr.html#\\*funcinexpr](#)

alter-table-stmt:

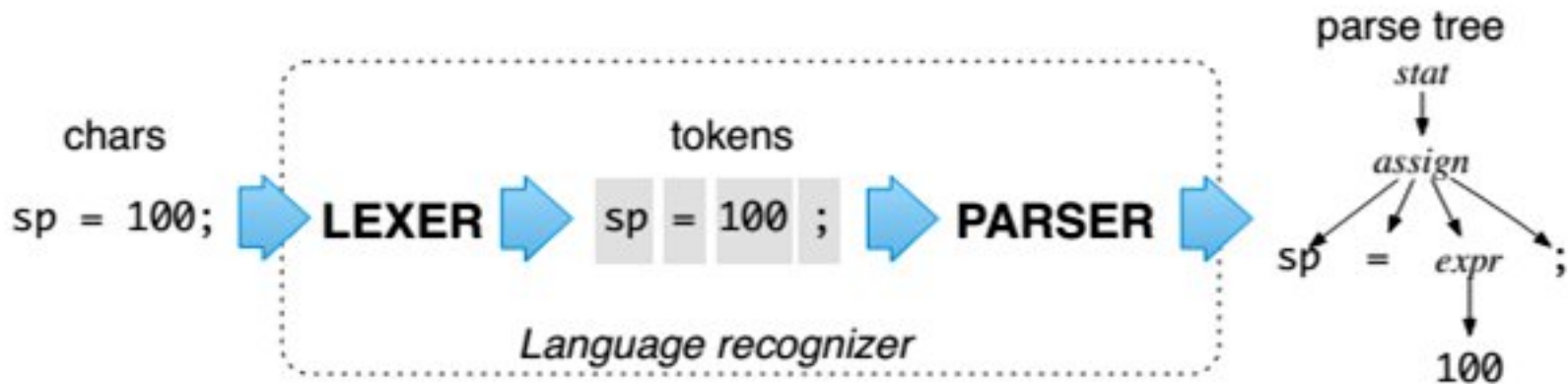


# **Синтаксический анализ текста программы**

# Анализ синтаксиса программы

**Лексер** – разбирает текст и представляет его в виде массива токенов. **Токен** – совокупность значения лексемы (значимого слова), ее типа и другой метайнформации.

**Парсер** – группирует по поступающие из лексера токены в дерево разбора, отражающее иерархию элементов программы и связи между ними.

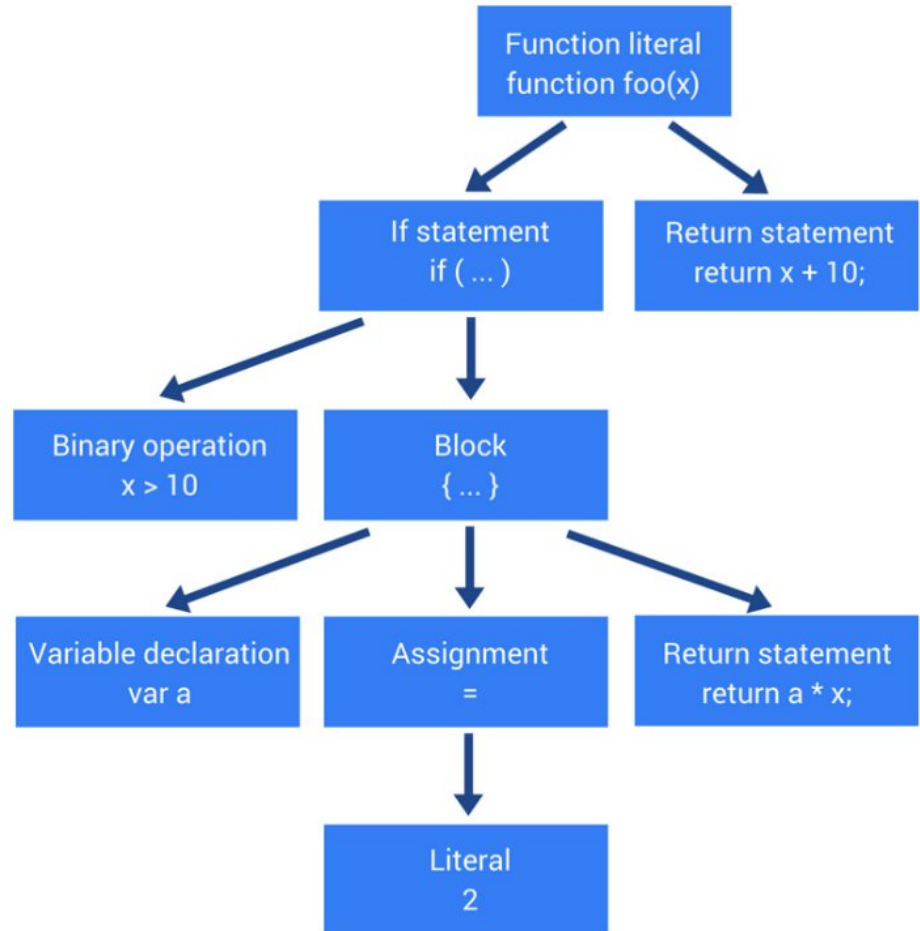


# Анализ синтаксиса программы

**Абстрактное синтаксическое дерево (AST)** — дерево разбора, из которого удалены незначимые токены (скобки, запятые, ...).

Это структурное представление исходного кода в виде дерева, где каждая вершина обозначает различные типы конструкций языка (выражение, переменную, оператор и т.п.)

```
function foo(x) {  
  if (x > 10) {  
    var a = 2;  
    return a * x;  
  }  
  
  return x + 10;  
}
```



# Абстрактное синтаксическое дерево

---

AST является универсальным, потому что:

1. **Независим от конкретного синтаксиса** — один и тот же код на разных языках (например, Python и JavaScript) может породить схожие AST-структуры при одинаковой семантике.
2. **Легко обрабатывается машиной** — AST упрощает анализ, трансформацию и генерацию кода, что делает его основой для компиляторов, интерпретаторов, линтеров, рефакторинг-инструментов и транспайлеров (например, Babel, TypeScript).
3. **Позволяет выполнять статический анализ** — можно выявлять ошибки, не запуская программу, например, неиспользуемые переменные или потенциальные уязвимости.
4. **Поддерживает трансформации** — через манипуляции с AST можно автоматически изменять код: оптимизировать, мигрировать между версиями языка, добавлять логирование и т.д.

## Особенности AST для SQL-запросов:

### 1. Высокоуровневая структура запроса

Узлы AST отражают основные компоненты SQL-запроса: `SELECT`, `FROM`, `WHERE`, `JOIN`, `GROUP BY`, `ORDER BY`, `HAVING`, `UNION` и т.д. Каждый из них представлен в виде узла, содержащего поддеревья для выражений, таблиц, условий и т.п.

### 2. Операторы и выражения

SQL-выражения (например, `col1 > 10`, `COUNT(*)`, `UPPER(name)`) также представлены в виде узлов. Это позволяет легко анализировать и трансформировать части запроса, например, для оптимизации или проверки безопасности (SQL-инъекции).

### 3. Подзапросы

Подзапросы (вложенные `SELECT`) становятся отдельными AST-поддеревьями, встроенными в родительское дерево.

## 4. Оптимизация на уровне AST

Многие СУБД используют AST как промежуточное представление для оптимизации запросов: перестановка условий, исключение дубликатов, выбор стратегии соединения и т.д.

## 5. Унификация запросов

AST позволяет привести различные синтаксические формы одного и того же запроса к общей структуре, что удобно для анализа и сравнения.

Таким образом, AST для SQL — это мощный инструмент для анализа, модификации и оптимизации запросов, адаптированный под декларативную природу языка.

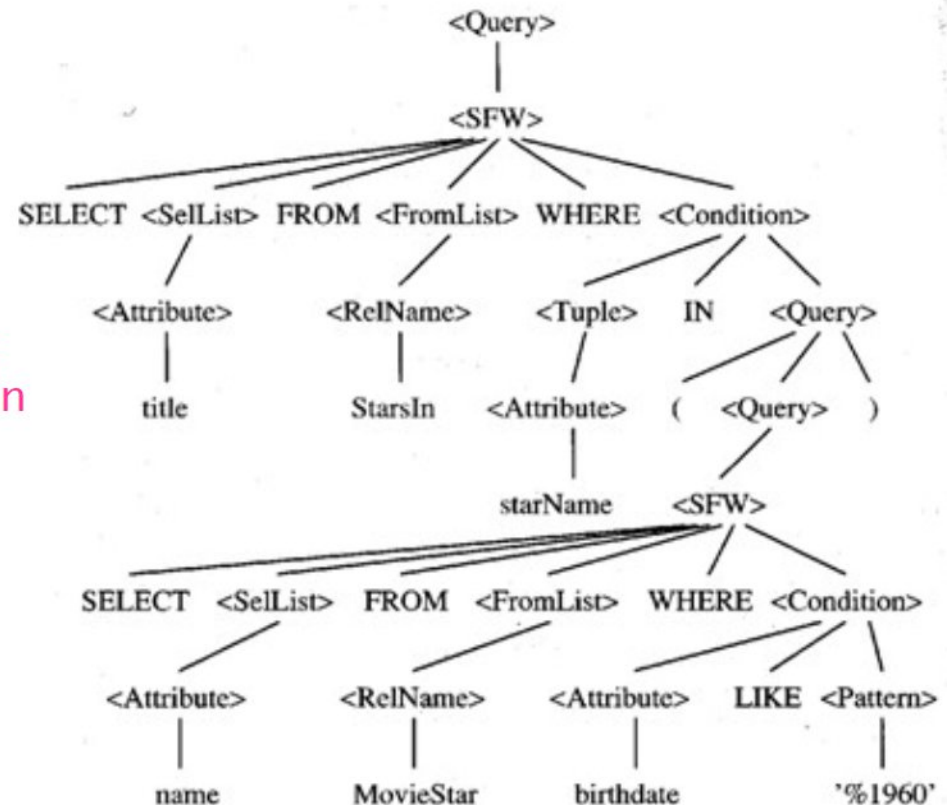
```
StarsIn(
  title, year, starName
)
```

```
MovieStar(
  name, address, gender, bdate
)
```

Query:

Give titles of movies that  
have at least one star born  
in 1960

```
SELECT title
FROM StarsIn
WHERE starName IN (
  SELECT name
  FROM MovieStar
  WHERE
    birthdate LIKE '%1960%'
);
```

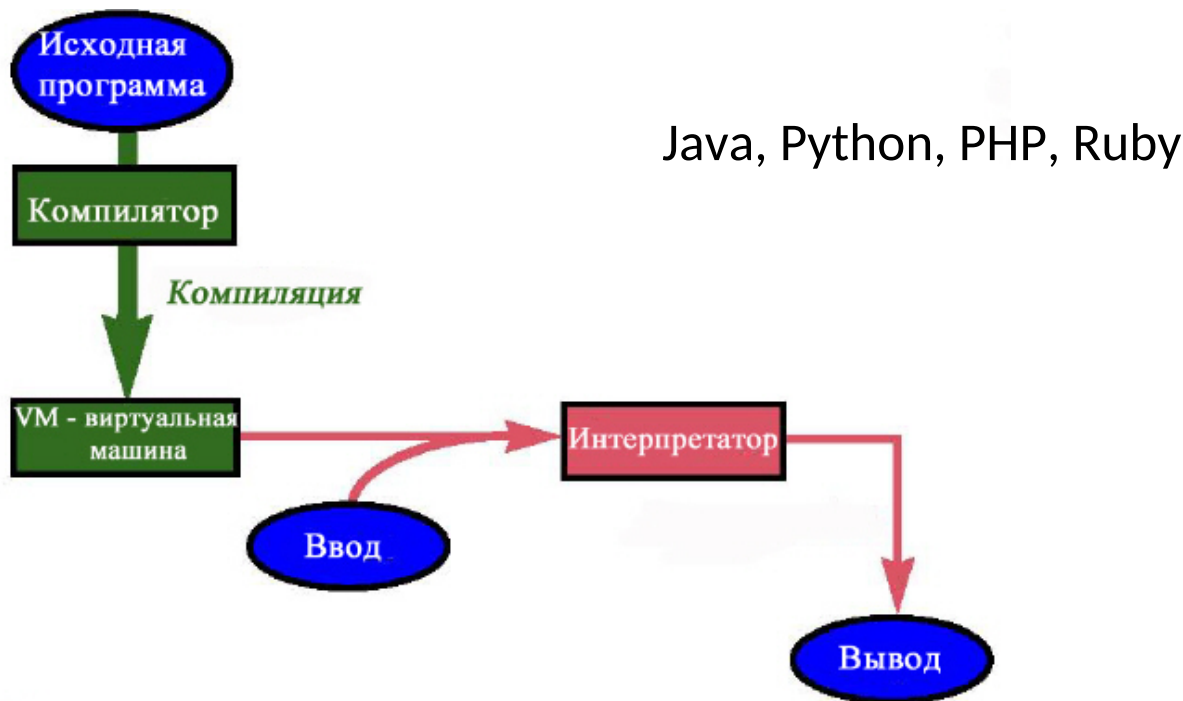


# **Шаги при компиляции программы на ЯП**

## Процесс компиляции

- 1. Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем (токенов).
- 2. Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.
- 3. Семантический анализ.** Дерево разбора обрабатывается для установления его семантики (смысла): привязка идентификаторов к определениям, проверка совместимости, определение типов выражений и т.д. Результат – промежуточное представление/код.
- 4. Оптимизация.** Удаление лишних конструкций и упрощение кода с сохранением его смысла.
- 5. Генерация машинного кода.**

# Комбинирование компиляции и интерпретации



Компилятор создает байт-код на промежуточном языке, понимаемом некоторой виртуальной машиной (интерпретатором байт-кода).

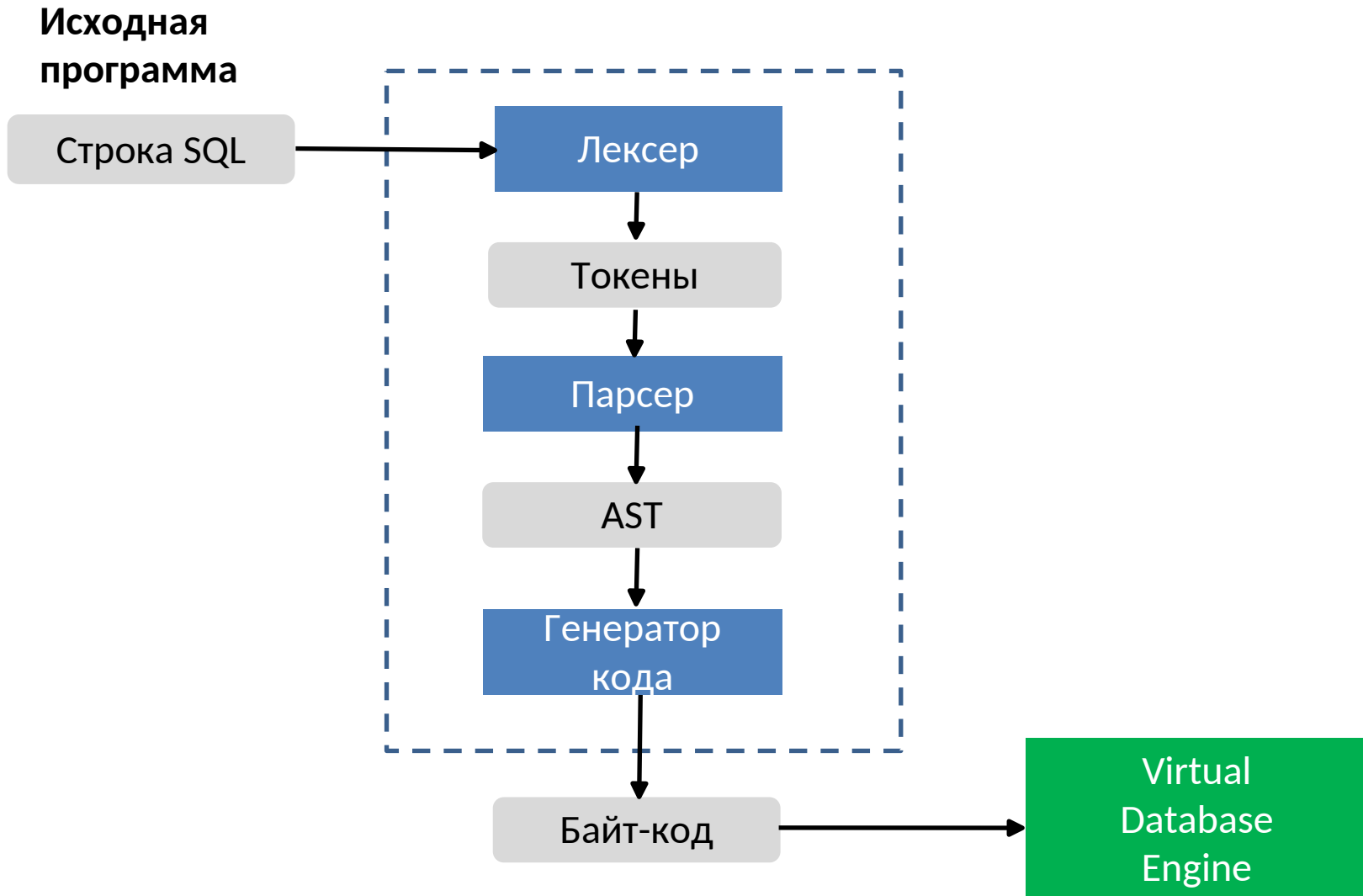
Преимущества:

- **Кроссплатформенность**, так как VM-код не зависит от специфики процессоров;
- **Повышение эффективности**, так как создаваемый промежуточный код легко интерпретируется.

# Компиляция SQL-запроса

# SQL - язык программирования

---



**Как увидеть байт-код,  
в который превратился SQL-запрос?**

# Команда EXPLAIN

---

Команда `EXPLAIN` в SQLite позволяет увидеть, как будет выполняться SQL-запрос, но не выполняя его на самом деле. Она показывает внутренние инструкции виртуальной машины VDBE (Virtual Database Engine) — это байт-код, который SQLite генерирует из SQL-запроса и затем исполняет.

**EXPLAIN** нужна для:

- Анализа логики выполнения запроса.
- Понимания, какие операции будут выполнены, в каком порядке.
- Отладки производительности: например, выявления лишних сканирований или отсутствия использования индексов.
- Обучения: чтобы понять, как SQLite "думает".

“🔍 Это инструмент низкоуровневой диагностики.”

## 📌 Что возвращает EXPLAIN?

Таблицу с колонками:

КОЛОНКА	ОПИСАНИЕ
<code>addr</code>	Адрес инструкции (номер строки в программе).
<code>opcode</code>	Операция (например, <code>OpenRead</code> , <code>Next</code> , <code>ResultRow</code> ).
<code>p1</code> , <code>p2</code> , <code>p3</code> , <code>p4</code> , <code>p5</code>	Параметры операции.
<code>comment</code>	Комментарий (если включён).

Каждая строка — это одна VDBE-инструкция, аналог команды в ассемблере.

Каждая инструкция VDBE — это трёхадресная команда вида:

```
1 opcode P1 P2 P3 P4 P5 comment
```

где:

- `opcode` — операция (например, `OpenRead`, `Column`, `Gt`, `ResultRow`).
  - `P1–P5` — параметры (номера регистров, константы, смещения).
  - `comment` — поясняющий комментарий (в выводе `.explain`).
-

# Байт-код VDBE

Рассмотрим запрос:

```
sql
1 SELECT name FROM users WHERE age > 25;
```

После компиляции SQLite может сгенерировать такой байт-код (упрощённо):

1	addr	opcode	p1	p2	p3	p4	p5	comment
2	----	-----	----	----	----	-----	--	-----
3	0	Init	0	12	0			Start
4	1	OpenRead	0	2	0	2		root=2, tab=users
5	2	Rewind	0	11	0			goto 11 if empty
6	3	Column	0	1	1			r[1] = users.age
7	4	Gt	2	10	1			if r[1] > 25 goto 10
8	5	Goto	0	11				
9	6	Next	0	3	0			continue loop
10	10	Column	0	0	3			r[3] = users.name
11	11	ResultRow	3	1	0			output row
12	12	Halt	0	0	0			

# Байт-код VDBE

1	addr	opcode	p1	p2	p3	p4	p5	comment
2	----	-----	---	---	---	---	---	-----
3	0	Init	0	12	0			Start
4	1	OpenRead	0	2	0	2		root=2, tab=users
5	2	Rewind	0	11	0			goto 11 if empty
6	3	Column	0	1	1			r[1] = users.age
7	4	Gt	2	10	1			if r[1] > 25 goto 10
8	5	Goto	0	11				
9	6	Next	0	3	0			continue loop
10	10	Column	0	0	3			r[3] = users.name
11	11	ResultRow	3	1	0			output row
12	12	Halt	0	0	0			

## Пояснение:

- `OpenRead` открывает таблицу `users`.
- `Rewind` и `Next` реализуют цикл по строкам.
- `Column` загружает значение столбца в регистр.
- `Gt` сравнивает `age` с константой `25`.
- `ResultRow` возвращает строку результата.

- [PRAGMA vdbe\\_trace](#)

## 4. The Opcodes

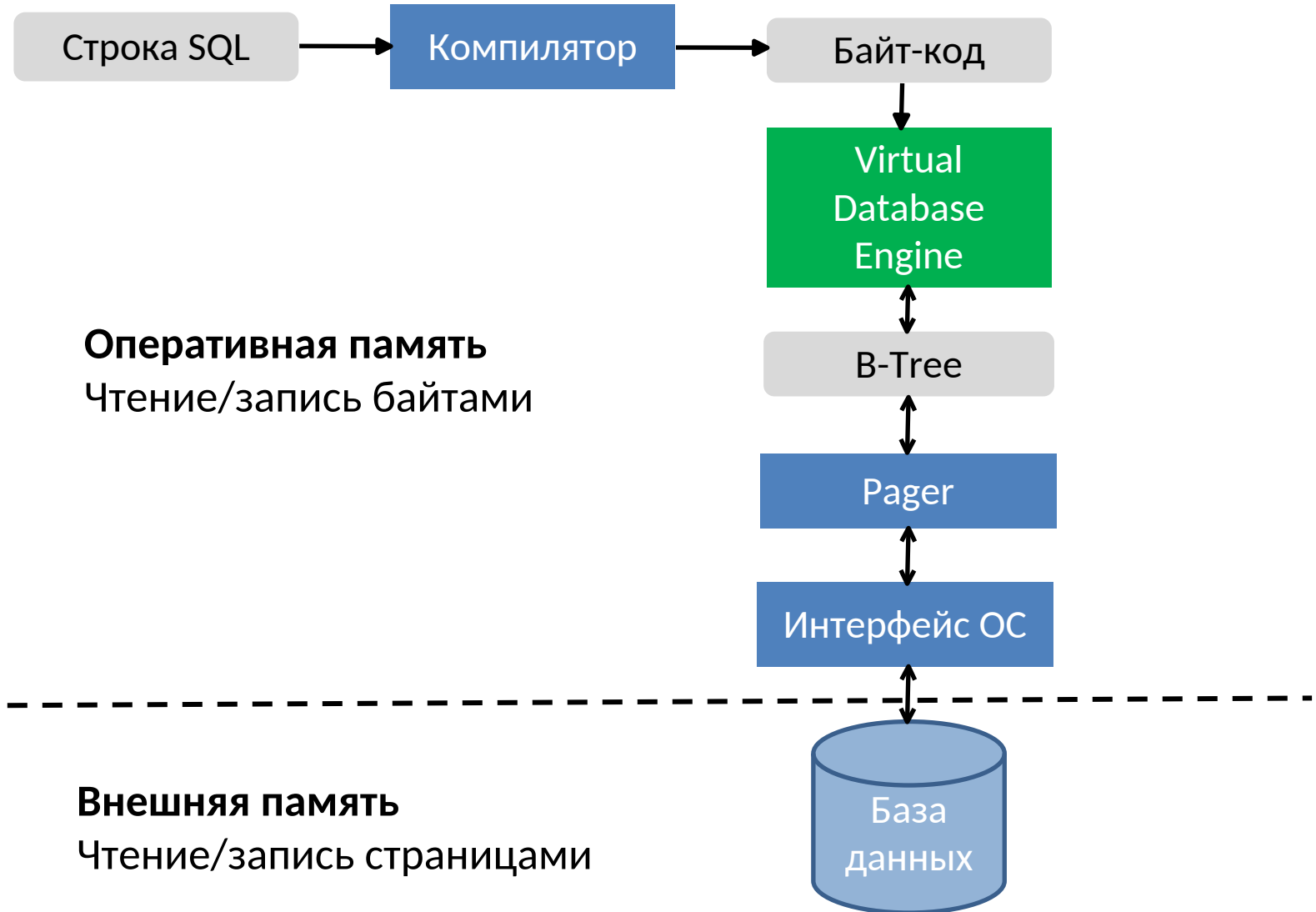
There are currently 191 opcodes defined by the virtual machine. All currently defined opcodes are described in the table below. This table was generated automatically by scanning the source code from the file [vdbe.c](#).

Remember: The VDBE opcodes are not part of the interface definition for SQLite. The number of opcodes and their names and meanings change from one release of SQLite to the next. The opcodes shown in the table below are valid for SQLite version 3.51.0 check-in [fb2c931ae597f](#) dated 2025-11-04.

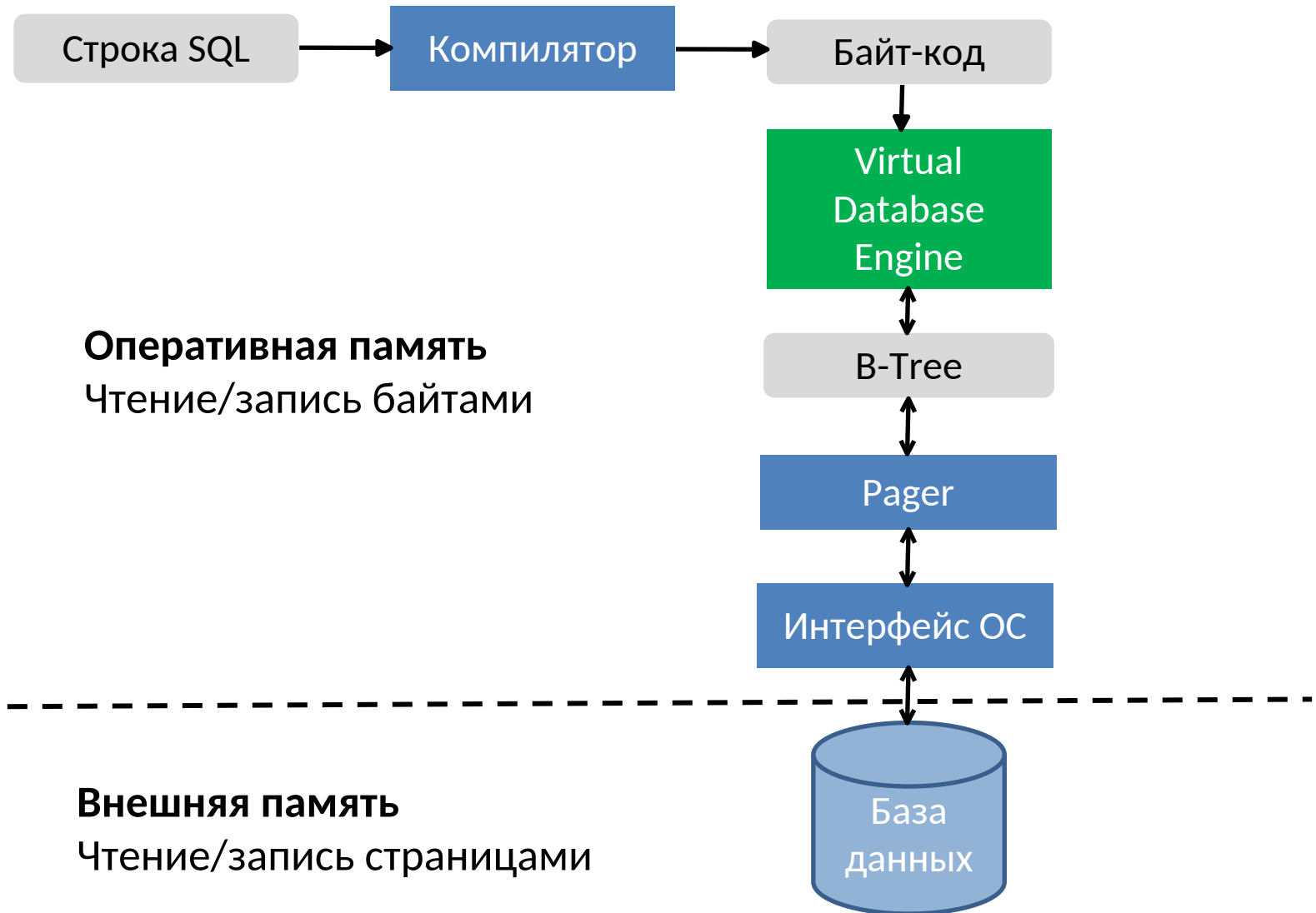
Opcode Name	Description
Abortable	Verify that an Abort can happen. Assert if an Abort at this point might cause database corruption. This opcode only appears in debugging builds.  An Abort is safe if either there have been no writes, or if there is an active statement journal.
Add	Add the value in register P1 to the value in register P2 and store the result in register P3. If either input is NULL, the result is NULL.
AddImm	Add the constant P2 to the value in register P1. The result is always an integer.  To force any register to be an integer, just add 0.

# **Архитектура СУБД и скорость выполнения SQL-запросов**

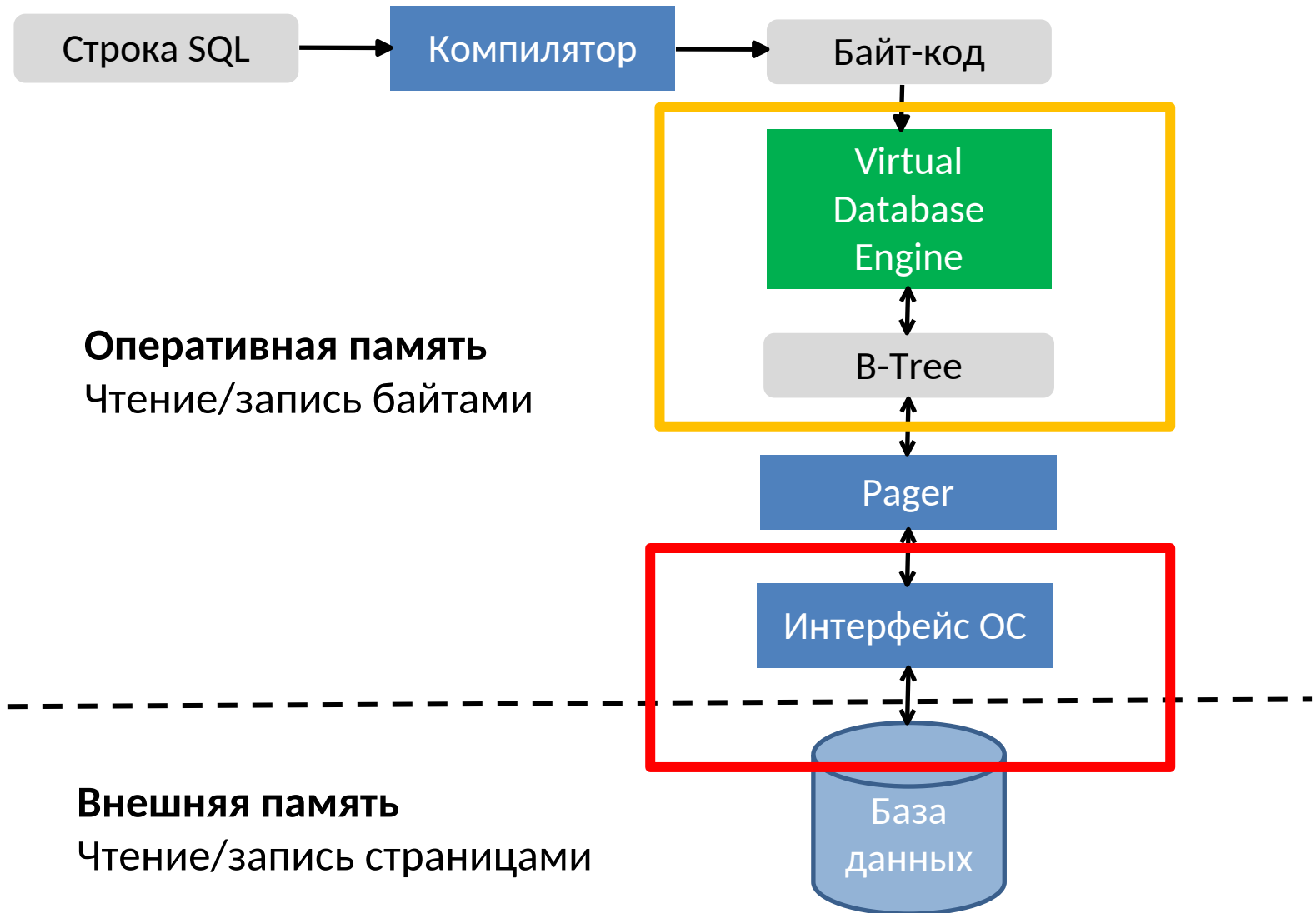
# Архитектура SQLite



# От чего зависит скорость выполнения запросов?



# От чего зависит скорость выполнения запросов?



# **Два фактора, влияющих на производительность запросов**

# Производительность запросов

## ✓ 1. Количество обращений к внешнему носителю (I/O-операции)

Это один из самых критичных факторов производительности в СУБД, особенно при работе с большими объёмами данных.

### Почему это важно:

- Чтение с диска (HDD/SSD) в тысячи раз медленнее, чем доступ к оперативной памяти.
- Каждое обращение к странице данных на диске — это дорогостоящая операция.
- Цель оптимизатора: минимизировать количество чтений с диска.

# Скорость типичных операций

Стоимость операции	нс (ns)	мкс ( $\mu$ s)	мс (ms)
Получение значения из L1	0.5		
Ошибка предсказания перехода в CPU	5		
Получение значения из L2	7		
Mutex lock/unlock	25		
Получение значения из RAM	100		
Сжатие 1Кб методом Zipru	3 000	3	
Отправка 1Кб через 1Гбит/сек сеть	10 000	10	
Чтение 4Кб с SSD (случайный доступ)	150 000	150	
<b>Чтение 1Мб из RAM (последовательный доступ)</b>	<b>250 000</b>	<b>250</b>	
Round trip внутри одного датацентра	500 000	500	
<b>Чтение 1Мб из SSD (последовательный доступ)</b>	<b>1 000 000</b>	<b>1 000</b>	<b>1</b>
Позиционирование HDD	10 000 000	10 000	10
<b>Чтение 1Мб из HDD (последовательный доступ)</b>	<b>20 000 000</b>	<b>20 000</b>	<b>20</b>
Round trip между США и Нидерландами	150 000 000	150 000	150

# Производительность запросов

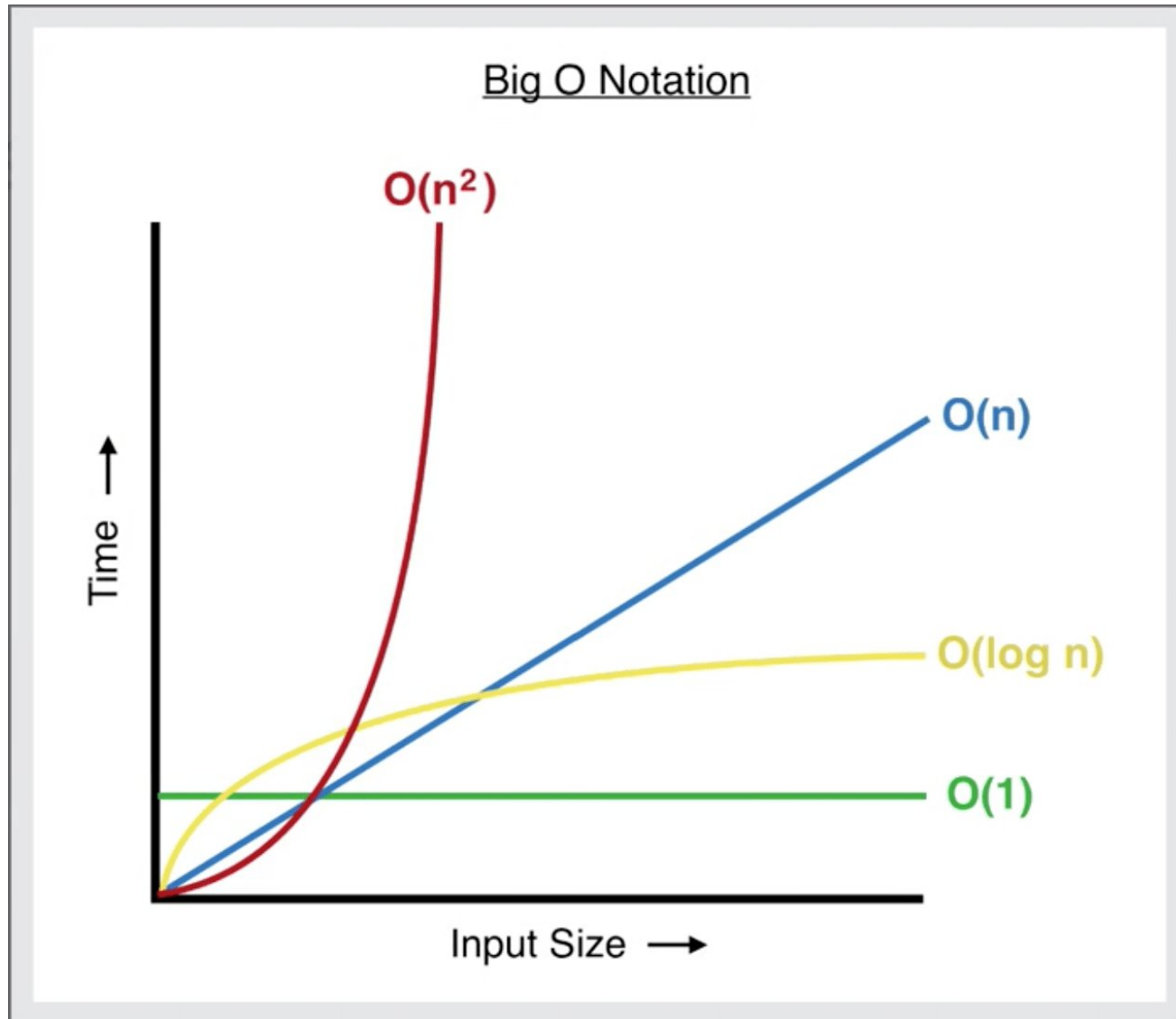
## ✓ 2. Алгоритмическая сложность алгоритма выполнения

Под этим подразумевается **вычислительная эффективность операций**, выполняемых над данными, уже загруженными в память.

### Почему это важно:

- Даже если данные в памяти, **неэффективный алгоритм** может тормозить выполнение.
- Оптимизатор SQL выбирает **план выполнения** с наименьшей оценкой стоимости (включая CPU и размер промежуточных результатов).

# Алгоритмическая сложность, время обработки



# Алгоритмическая сложность, время обработки

---

Если компьютер выполняет миллиард операций в секунду ...

$O()$	Миллиард элементов	Триллион элементов
$O(n)$	1 секунда	16 минут
$O(\log_2 n)$	30 наносекунд	40 наносекунд
$O(n \cdot \log_2 n)$	30 секунд	11 часов
$O(n^2)$	32 года	32 миллиона лет

**Две проблемы - одно решение.  
Индексы в базе данных**

# Индекс (алфавитный указатель) в книге

---

## —Index—

### —A—

about the author 128, 132, 412  
account info 295  
active table of contents 34, 120-124, 238-239,  
285-286, 354, 366, 370  
ACX 465-467  
Adobe 506  
advertising 434, 439-449  
age 312  
aggregator 17-18, 322  
alignment 68, 101-103, 105-106, 229-230, 261-262, 353-  
354, 380, 389  
Alt codes 39  
Amazon Associates 415  
Amazon Follow 430, 437, 480  
Amazon Giveaway 436-439  
Amazon Marketing Services (AMS) 439-449  
Android 167-169, 171, 371-375  
apostrophe 40, 42-44  
app 141-142  
Apple 169, 342, 372, 506

automatic renewal 327-329, 341, 343  
Automatically Update 73-75, 94, 144  
AZK 371

### —B—

back matter 124-129  
background 47, 93, 181, 184, 192-193, 246, 252-253, 355,  
370, 385, 390  
bank information 295  
Barnes & Noble 506  
biography 128, 132, 410  
black 47, 93, 184, 192, 252-253, 355, 370, 385, 390  
Blackberry 372-373  
blank line 27-28, 110, 112-114, 276-277, 284-285, 385  
blank page 354, 385-386  
block indent 50, 52, 67, 82, 106-107, 234-235  
blog 411, 429, 479  
Blogger 429  
bloggers 327, 430  
blurb 300-306, 364, 406, 411-412, 417, 477  
blurry 162-164, 172, 175, 193, 246, 387, 389  
body text 66, 68, 79-82, 92-94, 115, 233-235

# Индексирование

---

## Основные данные

Id	Фамилия	Имя	Возраст
1	Петров	Иван	20
2	Сидоров	Андрей	34
5	Иванова	Ольга	19
7	Антонов	Борис	43
11	Ливанов	Антон	25
15	Козлов	Сергей	26

### Поиск по фамилии

Сколько записей нужно  
прочитать?

# Индексирование

---

## Основные данные

Id	Фамилия	Имя	Возраст
1	Петров	Иван	20
2	Сидоров	Андрей	34
5	Иванова	Ольга	19
7	Антонов	Борис	43
11	Ливанов	Антон	25
15	Козлов	Сергей	26

Поиск по фамилии

$O(N)$  операций

# Индексирование

= Логическое упорядочение физических записей

## Основные данные

Id	Фамилия	Имя	Возраст
1	Петров	Иван	20
2	Сидоров	Андрей	34
5	Иванова	Ольга	19
7	Антонов	Борис	43
11	Ливанов	Антон	25
15	Козлов	Сергей	26

**Поиск по фамилии**

**$O(N)$  операций**

## Индекс

ID	Фамилия
7	Антонов
5	Иванова
15	Козлов
11	Ливанов
1	Петров
2	Сидоров

**Поиск по фамилии**

Сколько записей нужно прочитать?

# Индексирование

= Логическое упорядочение физических записей

## Основные данные

Id	Фамилия	Имя	Возраст
1	Петров	Иван	20
2	Сидоров	Андрей	34
5	Иванова	Ольга	19
7	Антонов	Борис	43
11	Ливанов	Антон	25
15	Козлов	Сергей	26

Поиск по фамилии  
 $O(N)$  операций

## Индекс

ID	Фамилия
7	Антонов
5	Иванова
15	Козлов
11	Ливанов
1	Петров
2	Сидоров

Поиск по фамилии  
 $O(\log_2 N)$  операций

# Индексирование

= Логическое упорядочение физических записей

Основные данные

Id	Фамилия	Имя	Возраст
1	Петров	Иван	20
2	Сидоров	Андрей	34
5	Иванова	Ольга	19
7	Антонов	Борис	43
11	Ливанов	Антон	25
15	Козлов	Сергей	26

Индекс

ID	Фамилия
7	Антонов
5	Иванова
15	Козлов
11	Ливанов
1	Петров
2	Сидоров

## Поиск в индексированной таблице:

1. Дихотомический поиск в индексе
2. Определение Id записи в основной таблице
3. Дихотомический поиск по Id в основной таблице

# Индексирование

= Логическое упорядочение физических записей

## Основные данные

N	Фамилия (PK)	Имя	Возраст
1	Петров	Иван	20
2	Сидоров	Андрей	34
3	Иванова	Ольга	19
4	Антонов	Борис	43
5	Ливанов	Антон	25
6	Козлов	Сергей	26

## Индекс

Фамилия (FK)	Номер записи
Антонов	4
Иванова	3
Козлов	6
Ливанов	5
Петров	1
Сидоров	2

## Поиск в индексированной таблице:

1. Дихотомический поиск в индексе
2. Определение номера записи в основной таблице
3. Считывание нужной записи в основной таблице

## Индексированная таблица

---

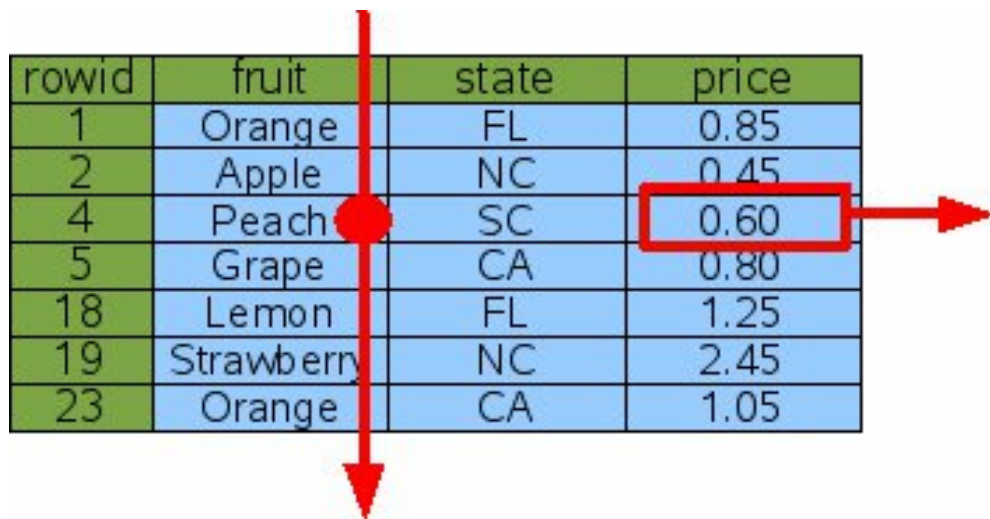
**Важно:** в идеале индексный файл имеет малый размер и полностью считывается в оперативную память за один раз.

# **Поиск строк в индексированной таблице**

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Логическое представление таблицы fruits

**SELECT price FROM fruits WHERE fruit='Peach';**



rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Сканирование по всей таблице => N операций

**SELECT price FROM fruits WHERE rowid=4;**

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Поиск по rowid =>  $\log_2 N$  операций

## Одноколоночный индекс

```
CREATE INDEX Idx1 ON fruits(fruit);
```

fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

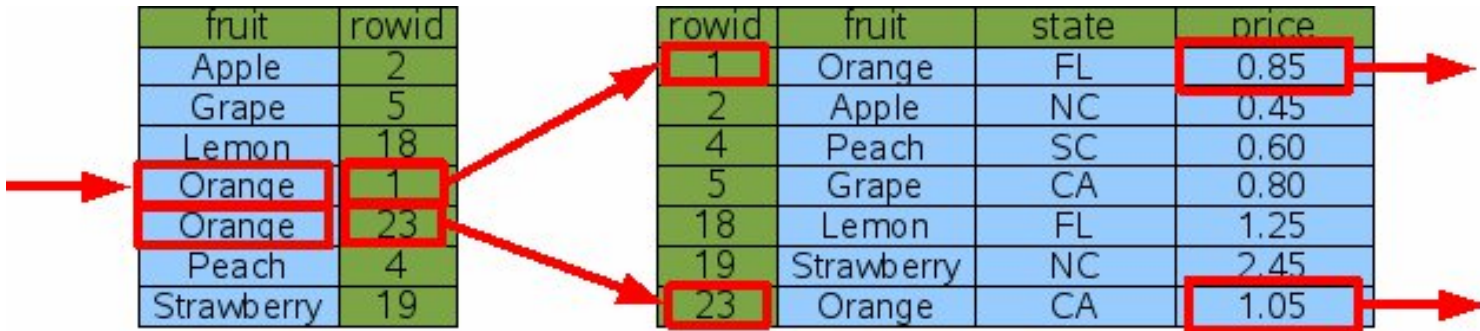
Логическое представление индекса по столбцу fruit

**SELECT price FROM fruits WHERE fruit='Peach';**



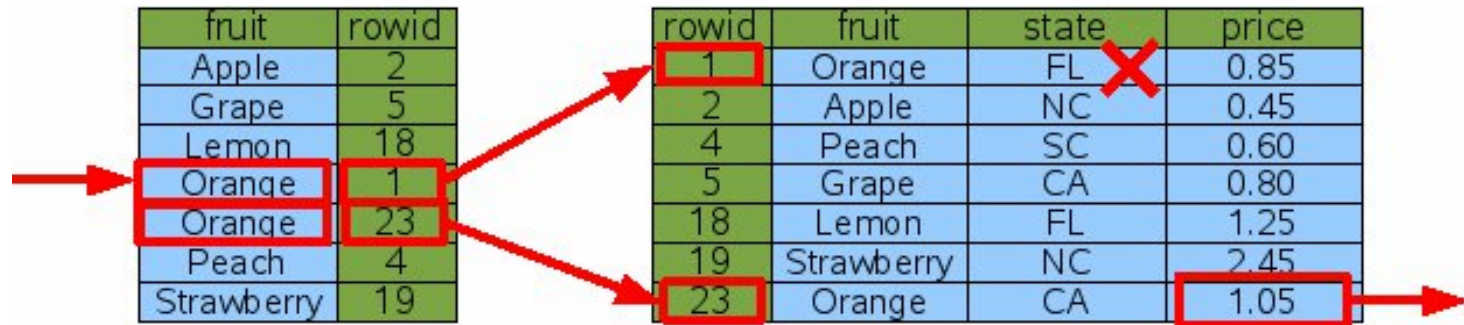
Поиск по индексу, один результат  $\Rightarrow 2 * \log_2 N$  операций

**SELECT price FROM fruits WHERE fruit='Orange';**



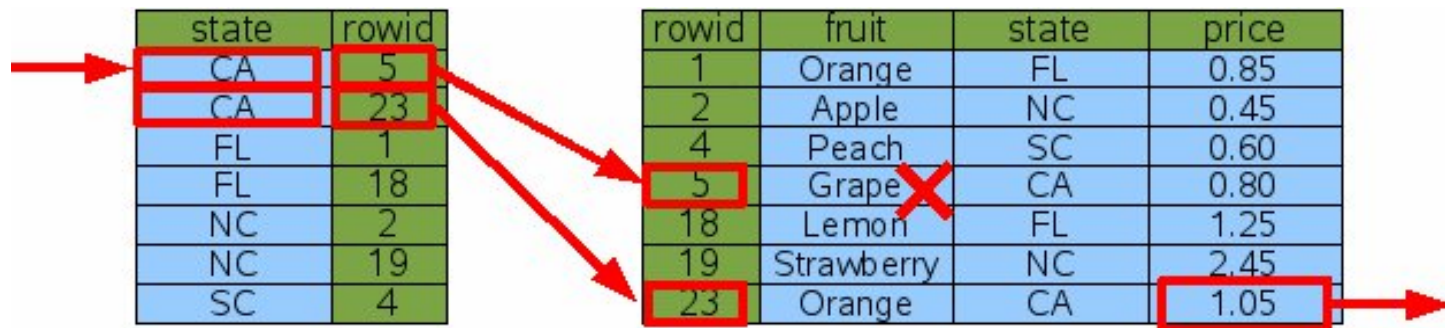
Несколько строк в результате  $\Rightarrow (K+1) \cdot \log_2 N$  операций

**SELECT price FROM fruits WHERE fruit='Orange' AND state='CA';**



Условие AND в WHERE

**CREATE INDEX Idx2 ON fruits(state);**



## Составной индекс

**CREATE INDEX Idx3 ON Fruits(fruit, state);**

fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

**SELECT price FROM fruits WHERE fruit='Orange' AND state='CA';**

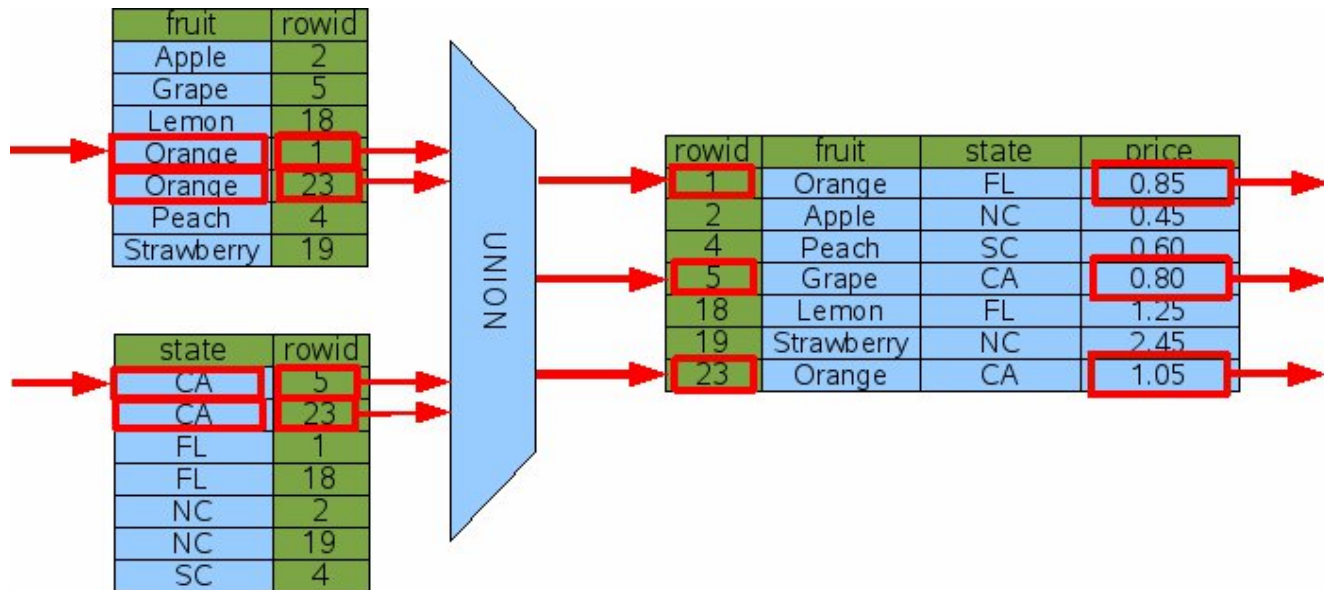
fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

$2 * \log_2 N$  операций

# Условие OR в WHERE

**SELECT price FROM Fruits WHERE fruit='Orange' OR state='CA';**



## Покрывающий индекс

**CREATE INDEX Idx4 ON Fruits(fruit, state, price);**

fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

**SELECT price FROM fruits WHERE fruit='Orange' AND state='CA';**

fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

$\log_2 N$  операций

## Преимущества покрывающего индекса

### 1. Высокая производительность

Исключается дорогостоящая операция чтения строк из таблицы (особенно если таблица широкая или физически фрагментирована).

### 2. Меньшее количество операций ввода-вывода

Индекс обычно компактнее таблицы (особенно если включает только нужные столбцы), поэтому его быстрее читать с диска или из кэша.

### 3. Эффективное использование кэша

Покрывающие индексы часто полностью помещаются в память, особенно при частых запросах к небольшому набору столбцов.

# Кластеризованный индекс

Кластеризованный индекс (clustered index) — это особый тип индекса, при котором физический порядок хранения строк в таблице совпадает с порядком значений индекса. Другими словами, данные таблицы отсортированы и хранятся непосредственно в структуре индекса (обычно в виде сбалансированного дерева — B-дерева).

---

## Почему он так называется?

Название «кластеризованный» происходит от слова «кластер» — то есть данные сгруппированы (кластеризованы) в физическом порядке, соответствующем ключу индекса.

Например, если кластеризованный индекс построен по столбцу `user_id`, то строки в файле данных будут физически расположены в порядке возрастания `user_id`.

# Кластеризованный индекс

## Основные особенности

1. В таблице может быть только один кластеризованный индекс  
Потому что данные могут быть физически упорядочены только одним способом.
2. Кластеризованный индекс определяет физическое расположение данных  
В отличие от некластеризованных индексов, которые хранят указатели на строки.
3. Часто (но не всегда) строится на первичном ключе  
Например, в SQL Server и MySQL (InnoDB) первичный ключ автоматически становится кластеризованным индексом, если пользователь не указал иное.

# Преимущества кластеризованного индекса

## 1. Быстрый диапазонный поиск

Запросы вида

```
sql
1 SELECT * FROM orders WHERE order_date BETWEEN '2025-01-01' AND '2025-12-31';
```

выполняются очень быстро, так как нужные строки физически лежат рядом на диске → минимизируется количество операций чтения.

## 2. Отсутствие дополнительного обращения к данным

Поскольку сами данные хранятся в листьях индекса, СУБД не нужна дополнительная операция (lookup) для извлечения полной строки — в отличие от некластеризованного индекса.

## 3. Эффективная сортировка и группировка

Если данные уже хранятся в нужном порядке, запросы с `ORDER BY` или `GROUP BY` по ключу индекса выполняются без сортировки в памяти.

## 4. Более эффективное использование кэша

Последовательное чтение физически смежных строк (например, при сканировании диапазона) лучше использует буферный пул или кэш диска.

# Таблица с Rowid в SQLite – аналог кластерного индекса

`rowid` — это скрытый 64-битный целочисленный столбец, который автоматически добавляется к каждой таблице, если она не является таблицей без `rowid` ( `WITHOUT ROWID` ).

Особенности `rowid` :

- Уникален для каждой строки в таблице.
- Автоматически увеличивается при вставке новых строк (если не задан явно).
- Используется как **физический ключ хранения**: данные в таблице упорядочены по `rowid` .
- Доступен во всех запросах, даже если вы его не объявляли:

```
sql
1 SELECT rowid, name FROM users;
```

Если вы объявляете столбец как `INTEGER PRIMARY KEY`, он становится псевдонимом для `rowid` :

```
sql
1 CREATE TABLE users (
2     id INTEGER PRIMARY KEY, -- это псевдоним rowid
3     name TEXT
4 );
```

# Таблица с Rowid в SQLite – аналог кластерного индекса

---

Да, по сути — вся обычная таблица в SQLite хранится как кластеризованное B-дерево по `rowid`.

Это означает:

- Данные физически упорядочены по `rowid`.
- Поиск по `rowid` (или по `INTEGER PRIMARY KEY`) выполняется за  $O(\log n)$  без дополнительного обращения к данным.
- Это аналог кластеризованного индекса в других СУБД, хотя термин «clustered index» в документации SQLite не используется.

# Сортировка строк в таблице

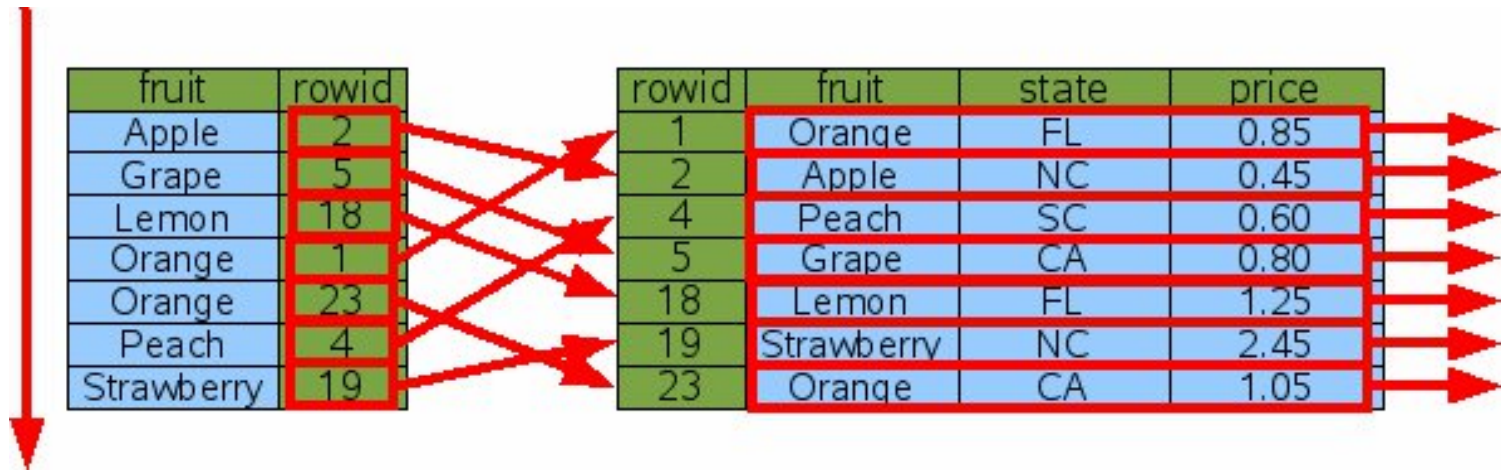
**SELECT \* FROM fruits ORDER BY fruit;**



Сортировка без индекса =>  $N \cdot \log_2 N$  операций

Также требуются временные хранилища для сортировки (b-tree)

**SELECT \* FROM fruits ORDER BY fruit;**



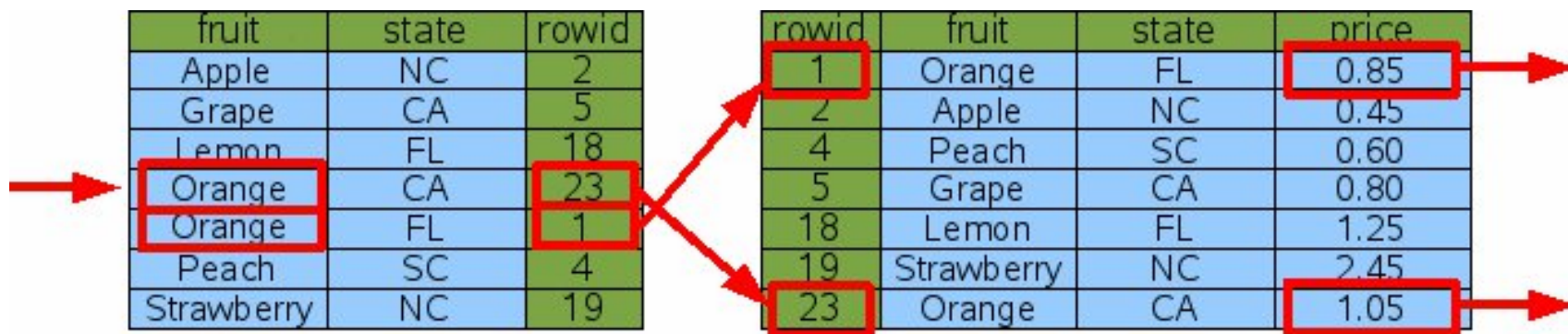
Сортировка с индексом =>  $N * \log_2 N$  операций

**SELECT fruits, state, price FROM fruits ORDER BY fruit;**

fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	10
Orange	CA	1.05	20
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	15

Сортировка с покрывающим индексом => N операций

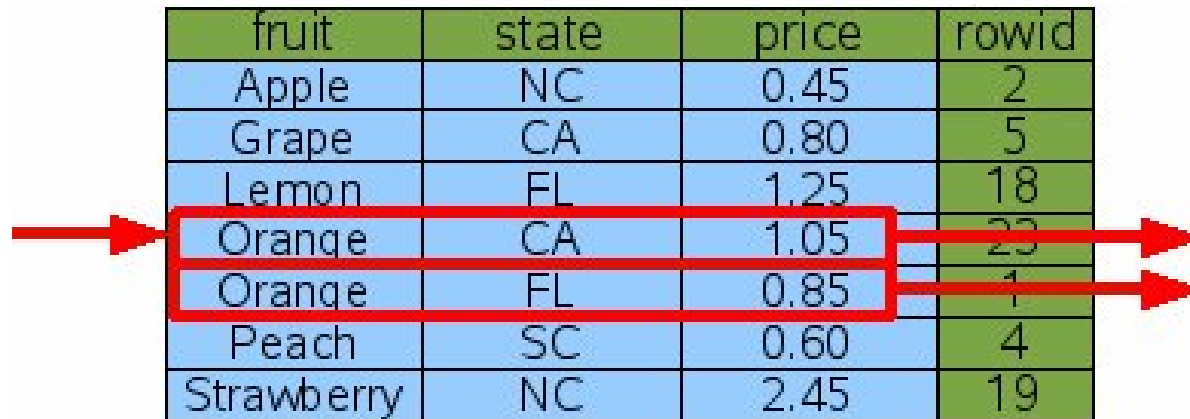
**SELECT price FROM fruits WHERE fruit='Orange' ORDER BY state**



Поиск и сортировка с составным индексом =>  $O(\log_2 N)$  операций

**SELECT price FROM fruits WHERE fruit='Orange' ORDER BY state**

fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19



Поиск и сортировка с покрывающим индексом

**Когда индексы могут помочь?**

# Назначение индексов

Индексы в реляционной базе данных нужны для ускорения операций поиска, сортировки, соединения и группировки данных, а также для обеспечения ограничений целостности (например, уникальности).

Вот подробнее, какие операции они ускоряют:

## ✓ 1. Поиск строк (фильтрация по условиям WHERE)

Индексы позволяют быстро находить строки, соответствующие условию, без полного сканирования таблицы.

Пример:

```
sql
1 SELECT * FROM users WHERE email = 'user@example.com';
```

Если на поле `email` есть индекс, SQLite может сразу найти нужную строку (или строки), используя дерево B-Tree (или другой тип структуры), а не просматривая всю таблицу.

# Назначение индексов

## ✓ 2. Сортировка (ORDER BY)

Если данные уже отсортированы в индексе — сортировка не требуется.

Пример:

```
sql
1 SELECT * FROM users ORDER BY created_at;
```

Если есть индекс по `created_at`, то данные можно просто прочесть в порядке индекса — это намного быстрее, чем сортировать миллионы строк в памяти.

## ✓ 3. Группировка (GROUP BY)

Индекс может ускорить группировку, особенно если поля группировки проиндексированы.

Пример:

```
sql
1 SELECT status, COUNT(*) FROM users GROUP BY status;
```

Если есть индекс по `status`, СУБД может эффективно пройти по всем значениям и сгруппировать их без дополнительной сортировки.

# Назначение индексов

## ✓ 4. Операции соединения (JOIN)

Индексы ускоряют поиск связанных строк при соединении таблиц.

Пример:

```
sql
1 SELECT * FROM orders o JOIN users u ON o.user_id = u.id;
```

Если на `u.id` и `o.user_id` есть индексы, соединение выполняется значительно быстрее.

# Назначение индексов

## ✓ 5. Проверка уникальности и первичные ключи

Индексы автоматически обеспечивают уникальность значений (в случае уникальных индексов или первичных ключей).

Пример:

```
sql
1 ✓ CREATE TABLE users (
2     id INTEGER PRIMARY KEY,
3     email TEXT UNIQUE
4 );
```

Здесь и `PRIMARY KEY`, и `UNIQUE` создают индексы, которые:

- Гарантируют отсутствие дубликатов.
- Ускоряют поиск по этим полям.

# Назначение индексов

## ✓ 6. Частичное покрытие запроса (Covering Index)

Если индекс содержит **все поля**, необходимые запросу, СУБД может получить данные **только из индекса**, не обращаясь к таблице.

Пример:

sql



```
1 CREATE INDEX idx_status_date ON orders (status, created_at);  
2 SELECT status, created_at FROM orders WHERE status = 'shipped';
```

Такой запрос может быть полностью удовлетворён индексом — это очень быстро.

**Всегда ли индексы полезны?**

## Плата за индексы

- Занимают дополнительное место на диске
- Могут замедлять операции модификации данных, так как при этом перестраивается индекс
- Накладные расходы на поддержание согласованности таблицы и индексов

Не стоит создавать индексы «на всякий случай», только осознанно, на основе анализа реальных запросов.

## × Что индексы НЕ ускоряют (или ускоряют слабо)

- `INSERT`, `UPDATE`, `DELETE` — наоборот, могут замедляться, потому что нужно обновлять и индексы тоже.
  - **Маленькие таблицы** — индексы не дают выигрыша, если таблица помещается в память.
  - **Запросы с функциями или выражениями** — например, `WHERE UPPER(name) = 'JOHN'` не использует обычный индекс по `name`, если нет функционального индекса.
  - **Запросы с подстановочными символами в начале** — например, `WHERE name LIKE '%ov'` не использует индекс эффективно.
-

## Селективность столбца

Селективность (Selectivity) — это отношение числа уникальных значений к общему числу строк в таблице:

$$\text{Селективность} = \frac{\text{Количество уникальных значений}}{\text{Общее количество строк}}$$

Иногда под селективностью понимают вероятность того, что запрос по одному значению этого столбца вернёт **одну конкретную строку** — то есть насколько хорошо столбец "отделяет" одну строку от других.

- Чем **выше** селективность → тем **лучше** столбец подходит для индекса.
- Селективность может быть близка к 1 (очень высокая) или к 0 (очень низкая).

## Селективность столбца

СТОЛБЕЦ	ЗНАЧЕНИЯ	КОЛ-ВО СТРОК	УНИКАЛЬНЫХ	СЕЛЕКТИВНОСТЬ
user_id	1, 2, 3, ..., 1 000 000	1 000 000	1 000 000	~1.0
email	разные email	500 000	500 000	~1.0
age	18–90	1 000 000	~73	~0.000073
gender	'M', 'F'	1 000 000	2	~0.000002

# Селективность столбца

Селективность напрямую влияет на эффективность использования индекса:

## 1. Высокая селективность → индекс полезен

- Например, поиск по `user_id = 12345` быстро находит одну строку.
- Индекс позволяет избежать полного сканирования таблицы.
- Оптимизатор запросов скорее выберет такой индекс.

## 2. Низкая селективность → индекс может не использоваться

- Если `gender = 'M'`, то 50% строк подходят.
- Чтение индекса + случайные обращения к таблице (random I/O) может быть медленнее, чем простое полное сканирование (sequential scan).
- В таких случаях оптимизатор часто игнорирует индекс.

# Селективность столбца

## 💡 Практические рекомендации

- Первичные ключи и уникальные поля — максимальная селективность → всегда индексируются.
- Столбцы с высокой селективностью (например, `email`, `passport_number`) — хорошие кандидаты на индекс.
- Столбцы с низкой селективностью (например, `status`, `gender`) — индексировать не всегда оправданно.
- Составные индексы могут повысить эффективность даже при низкой селективности отдельных полей.

# План выполнения SQL-запроса

`EXPLAIN QUERY PLAN` — это инструмент, который помогает понять, как СУБД планирует выполнение запроса:

- `SCAN` — полное сканирование (медленно).
- `SEARCH` — поиск с использованием индекса (быстро).
- `USE TEMP B-TREE FOR ...` — сортировка или группировка в памяти (может быть медленно при больших объёмах).

Это помогает оптимизировать запросы и структуру индексов.

#### Важно:

- `EXPLAIN QUERY PLAN` не выполняет запрос, а только показывает его план.
- Вывод не такой подробный, как `EXPLAIN` (который показывает байткод виртуальной машины SQLite), но более читаемый и ориентирован на пользователя.

# EXPLAIN QUERY PLAN

## Пример 1: Использование индекса

sql

```
1 EXPLAIN QUERY PLAN
2 SELECT * FROM users WHERE city = 'London';
```

Вывод:

```
1 0|0|0|SEARCH TABLE users USING INDEX idx_city_age (city=?) (~100 rows)
```

- Используется индекс `idx_city_age` для поиска по `city`.
- Это эффективно.

# EXPLAIN QUERY PLAN

## Пример 2: Полное сканирование

sql

```
1 ▾ EXPLAIN QUERY PLAN
2  SELECT * FROM users WHERE age = 25;
```

Вывод:

```
1  0|0|0|SCAN TABLE users (~50000 rows)
```

- Нет индекса по `age`, используется полное сканирование.
- Это неэффективно.

# EXPLAIN QUERY PLAN

## Пример 3: Сортировка с временным деревом

sql

```
1 ▾ EXPLAIN QUERY PLAN
2  SELECT * FROM users ORDER BY name;
```

Вывод:

```
1  0|0|0|SCAN TABLE users
2  0|0|0|USE TEMP B-TREE FOR ORDER BY
```

- Нет индекса по `name`, сортировка происходит в памяти с помощью временного B-Tree.

# EXPLAIN QUERY PLAN

## Пример 4: JOIN с индексами

sql



```
1 EXPLAIN QUERY PLAN
2 SELECT u.name, o.date
3 FROM users u
4 JOIN orders o ON u.id = o.user_id
5 WHERE u.city = 'Berlin';
```

Вывод может быть таким:



```
1 0|0|0|SEARCH TABLE users USING INDEX idx_city_age (city=?) (~10 rows)
2 0|1|1|SEARCH TABLE orders USING INDEX idx_orders_user_id (user_id=?) (~1 row)
```

- Обе таблицы используют индексы — эффективно.

# **Статистика о распределении данных в таблицах**

# Команда ANALYZE

Команда `ANALYZE` в SQLite собирает статистику о распределении данных в таблицах и индексах и сохраняет её в служебные системные таблицы (в основном — `sqlite_stat1`, а при использовании опций — также `sqlite_stat2`, `sqlite_stat3`, `sqlite_stat4`). Эта статистика используется оптимизатором запросов для выбора наиболее эффективного плана выполнения (например, какие индексы использовать, в каком порядке соединять таблицы и т.д.).

После выполнения `ANALYZE` SQLite «знает»:

- Сколько строк в таблице.
- Насколько селективен каждый индекс (например, сколько уникальных значений в столбце).
- Какие значения встречаются часто, а какие — редко.

# Команда ANALYZE

## 1. Общий анализ всей базы

```
sql
1 ANALYZE;
```

— собирает статистику по всем таблицам и индексам в базе.

## 2. Анализ конкретной таблицы

```
sql
1 ANALYZE users;
```

— собирает статистику только для таблицы `users` и всех её индексов.

## 3. Анализ конкретного индекса

```
sql
1 ANALYZE idx_user_email;
```

— обновляет статистику только для индекса `idx_user_email`.

# Команда ANALYZE

Допустим, у вас есть таблица:

```
sql
1 CREATE TABLE orders (
2   id INTEGER PRIMARY KEY,
3   user_id INTEGER,
4   status TEXT,
5   amount REAL
6 );
7
8 CREATE INDEX idx_status ON orders(status);
9 CREATE INDEX idx_user_id ON orders(user_id);
```

И вы выполняете запрос:

```
sql
1 SELECT * FROM orders WHERE status = 'completed';
```

- Если большинство заказов имеют статус `'completed'` (например, 95%), то использовать индекс `idx_status` невыгодно — быстрее просто прочитать всю таблицу.
- Если же `'completed'` — редкое значение (например, 1%), то индекс даст большой выигрыш.

Без `ANALYZE` SQLite может неправильно оценить селективность и выбрать не тот план.

# Команда ANALYZE

## Как посмотреть собранную статистику?

```
sql
```

```
1 SELECT * FROM sqlite_stat1;
```

## Пример результата:

```
1 tbl      | idx          | stat
2 -----|-----|-----
3 orders  | idx_status  | 100000 2
4 orders  |             | 100000
```

- **100000** — общее число строк в таблице.
- **2** — примерное число уникальных значений в столбце **status** (то есть высокая селективность → индекс полезен).

# Команда ANALYZE

Когда стоит выполнять `ANALYZE` ?

- После массовой загрузки данных (например, импорта).
- После значительного изменения данных (удаление/вставка миллионов строк).
- Если вы заметили, что запросы стали работать медленнее, несмотря на наличие индексов.
- Перед выполнением сложных аналитических запросов в приложениях с меняющимся объёмом данных.

“💡 SQLite не обновляет статистику автоматически, поэтому `ANALYZE` нужно вызывать вручную.”

# Оптимизатор запросов в SQLite

# Оптимизация запросов

---

Оптимизатор запросов в SQLite принимает решение об использовании индекса на основе статической оценки стоимости (cost-based optimization), но в сильно упрощённой форме по сравнению с крупными СУБД (например, PostgreSQL или Oracle). Он не использует сложные модели стоимости, а полагается на:

1. Наличие подходящего индекса
2. Статистику по данным (если собрана командой `ANALYZE` )
3. Правила селективности и структуры запроса

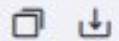
# Оптимизация запросов

## 1. Подходит ли индекс по структуре?

Оптимизатор сначала проверяет, может ли индекс использоваться для условия в `WHERE`, `ORDER BY`, `GROUP BY` или `JOIN`.

Пример:

sql



```
1 CREATE INDEX idx_user_status ON orders(user_id, status);
```

Запрос:

sql



```
1 SELECT * FROM orders WHERE user_id = 123 AND status = 'shipped';
```

→ индекс **подходит**, потому что оба столбца входят в начало составного индекса.

## 2. Оценка "полезности" индекса

Если индекс подходит, SQLite оценивает, даст ли он выигрыш.

Без `ANALYZE` :

- Считается, что все значения равномерно распределены.
- Для `=` — предполагается, что индекс уменьшит число строк в ~10 раз (грубая эвристика).
- Для `LIKE`, диапазонов (`BETWEEN`, `>`) — применяются другие приблизительные правила.

С `ANALYZE` :

- Используются данные из `sqlite_stat1` (и, при наличии, `sqlite_stat4`):
  - Общее число строк в таблице.
  - Число уникальных значений в столбце (это мера селективности).

# Оптимизация запросов

---

Пример:

- Если в `status` всего 2 уникальных значения ( `'pending'` , `'completed'` ), а строк — 1 000 000, то индекс по `status` почти бесполезен: каждое значение встречается ~500 000 раз.
- Оптимизатор может решить: **лучше сделать полное сканирование таблицы**, чем прыгать по индексу и читать половину строк по одной.

## 3. Сравнение стоимости планов

SQLite перебирает **все возможные индексы**, подходящие для запроса, и оценивает **примерную "стоимость"** каждого плана:

- Сколько, по оценке, строк будет прочитано?
- Нужно ли делать **дополнительный lookup** в таблицу (для некластеризованных индексов)?
- Нужна ли **сортировка** после сканирования?

Затем выбирается план с **наименьшей оценочной стоимостью**.

# Оптимизация запросов

Итог: как SQLite решает использовать индекс или нет?

ФАКТОР	ВЛИЯНИЕ НА РЕШЕНИЕ
Индекс покрывает условия <code>WHERE</code> ?	✓ Обязательное условие
Высокая селективность (редкое значение)?	✓ Скорее да
Низкая селективность (частое значение)?	× Скорее нет
Есть ли <code>ANALYZE</code> и статистика?	✓ Решение точнее
Требуется ли сортировка?	✓ Может повлиять на выбор индекса с <code>ORDER BY</code>
Таблица очень маленькая?	× Индекс может игнорироваться (накладные расходы > выгоды)

# Оптимизация запросов

---

## Советы по оптимизации

1. Выполняйте `ANALYZE` после массовых изменений данных.
2. Используйте `EXPLAIN QUERY PLAN`, чтобы проверять, как выполняются запросы.
3. Проектируйте составные индексы в порядке, соответствующем вашим запросам.
4. Избегайте индексов на низкоселективных столбцах (например, `is_active BOOLEAN`), если только не используете покрывающие индексы или фильтрацию по редкому значению (`is_active = 0` при 99% активных записей).

Таким образом, хотя оптимизатор SQLite прост, при правильном подходе он принимает весьма разумные решения.