

Реляционные базы данных: Индексы



Роман
Гордиенко



Роман Гордиенко

Backend Developer, Factory5



План занятия

1. [План запроса](#)
2. [Индексы](#)
3. [Типы индексов](#)
4. [Стоимость](#)
5. [Итоги](#)
6. [Домашнее задание](#)



План запроса

EXPLAIN

Оператор **EXPLAIN** демонстрирует этапы выполнения запроса и может быть использован для оптимизации.

По результату работы **EXPLAIN** можно выяснить, где в запросе узкие места, нужно ли использовать индексы, верный ли порядок и алгоритмы джойна выбраны при соединении таблиц и так далее.

EXPLAIN

EXPLAIN работает с **SELECT**, **DELETE**, **INSERT**, **REPLACE** и **UPDATE** операторами. В **MySQL 8.0.19** и более поздних версиях он также работает с оператором **TABLE**.

Оператор **EXPLAIN** выводит план запроса.

EXPLAIN

При использовании оператора **EXPLAIN** можно указать формат вывода с помощью оператора **FORMAT**:

- **TRADITIONAL** — вывод в табличном формате;
- **JSON** — вывод в формате **JSON**;
- **TREE** — древовидный вывод с более точными описаниями обработки запросов, чем **TRADITIONAL**.

EXPLAIN

```
EXPLAIN FORMAT = TRADITIONAL
SELECT *
FROM customer c
JOIN address a ON a.address_id = c.address_id
JOIN city c2 ON c2.city_id = a.city_id
WHERE c2.city_id = 17;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	c2		const	PRIMARY	PRIMARY	2	const	1	100.0	
1	SIMPLE	a		ref	PRIMARY,idx_fk_city_id	idx_fk_city_id	2	const	1	100.0	
1	SIMPLE	c		ref	idx_fk_address_id	idx_fk_address_id	2	sakila.a.address_id	1	100.0	

EXPLAIN ANALYZE

В MySQL 8.0.18 добавлена возможность использования оператора **EXPLAIN ANALYZE**, который запускает оператор и производит **EXPLAIN** вывод вместе с синхронизацией и дополнительной, основанной на итераторах, информацией о том, как ожидания оптимизатора совпадают с фактическим выполнением.

EXPLAIN ANALYZE всегда использует **TREE** формат.

EXPLAIN ANALYZE

Для каждого итератора предоставляется следующая информация:

- ориентировочная стоимость исполнения,
- расчетное количество возвращаемых строк,
- фактическое время возврата первой строки в миллисекундах,
- фактическое время возврата всех строк в миллисекундах (при наличии нескольких циклов этот пункт показывает среднее время на цикл),
- количество строк, возвращаемых итератором,
- количество циклов.

EXPLAIN ANALYZE

```
EXPLAIN ANALYZE
SELECT *
FROM customer c
JOIN address a ON a.address_id = c.address_id
JOIN city c2 ON c2.city_id = a.city_id
WHERE c2.city_id = 17;
```


-> Nested loop inner join (cost=1.68 rows=1) (actual time=0.088..0.092 rows=1 loops=1)

-> Index lookup on a using idx_fk_city_id (city_id=17) (cost=0.95 rows=1) (actual time=0.070..0.071 rows=1 loops=1)

-> Index lookup on c using idx_fk_address_id (address_id=a.address_id) (cost=0.72 rows=1) (actual time=0.016..0.018 rows=1 loops=1)



Индексы



INDEX – это инструмент, который
позволяет оптимизировать выборку из
базы данных, значительно сокращая время
на получение данных.

Индексы

Без индекса **MySQL** должен начать с первой строки, а затем прочитать всю таблицу, чтобы найти соответствующие строки. Чем больше таблица, тем больше это стоит.

Если таблица имеет индекс для рассматриваемых столбцов, **MySQL** может быстро определить позицию для поиска в середине файла данных, не просматривая все данные. Это намного быстрее, чем последовательное чтение каждой строки.

Индексы

Создадим временную таблицу и внесем в нее данные:

```
CREATE TABLE film_temp (  
    film_id INT,  
    title VARCHAR(50),  
    description TEXT,  
    language_id INT,  
    release_year INT  
);  
  
INSERT INTO film_temp  
SELECT film_id, title, description, language_id, release_year FROM film;
```

В данной таблице будут отсутствовать ограничения и индексы.

```
SELECT *  
FROM INFORMATION_SCHEMA.STATISTICS  
WHERE TABLE_NAME='film_temp';
```

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	NON_UNIQUE	INDEX_SCHEMA	INDEX_NAME	SEQ_IN_INDEX	COLUMN_NAME	COLLATION
---------------	--------------	------------	------------	--------------	------------	--------------	-------------	-----------

Индексы

Посмотрим на план запроса, где нужно получить фильм с id = 100:

```
EXPLAIN
SELECT *
FROM film_temp
WHERE film_id = 100;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film_temp		ALL					1000	10.0	Using where

Видим, что происходит обычное сканирование таблицы, отсутствует значение **possible_keys**:

```
EXPLAIN ANALYZE
SELECT *
FROM film_temp
WHERE film_id = 100;
-> Filter: (film_temp.film_id = 100) (cost=103.00 rows=100) (actual
time=0.186..1.624 rows=1 loops=1)
    -> Table scan on film_temp (cost=103.00 rows=1000) (actual
time=0.034..1.477 rows=1000 loops=1)
```


Индексы

Добавим на столбец `film_id` ограничения первичного ключа, которое включает индекс:

```
ALTER TABLE film_temp ADD PRIMARY KEY (film_id);
```

Посмотрим на результат плана запроса:

```
EXPLAIN
SELECT *
FROM film_temp
WHERE film_id = 100;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film_temp		const	PRIMARY	PRIMARY	4	const	1	100.0	

Результат **EXPLAIN ANALYZE** вернет:

```
-> Rows fetched before execution (cost=0.00..0.00 rows=1) (actual
time=0.000..0.001 rows=1 loops=1)
```

Индексы

Изменим немного данные и получим фильмы по двойному условию:

```
UPDATE film_temp  
SET release_year = 2005  
WHERE film_id <= 500;
```

```
EXPLAIN ANALYZE  
SELECT *  
FROM film_temp  
WHERE language_id = 1 AND release_year = 2006;  
-> Filter: ((film_temp.release_year = 2006) and (film_temp.language_id = 1))  
(cost=110.17 rows=10) (actual time=0.353..0.686 rows=500 loops=1)  
    -> Table scan on film_temp (cost=110.17 rows=1000) (actual  
time=0.025..0.567 rows=1000 loops=1)
```

Происходит обычный **table scan**.

Индексы

Создадим составной индекс:

```
CREATE INDEX lang_year ON film_temp(language_id, release_year);
```

Проверим, что индексы есть:

```
SELECT *  
FROM INFORMATION_SCHEMA.STATISTICS  
WHERE TABLE_NAME='film_temp';
```

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	NON_UNIQUE	INDEX_SCHEMA	INDEX_NAME	SEQ_IN_INDEX	COLUMN_NAME	COLLATION	CARDINALITY	SUB_PART	PACKED	NULLABLE	INDEX_TYPE
def	sakila	film_temp	1	sakila	lang_year	1	language_id	A	1			YES	BTREE
def	sakila	film_temp	1	sakila	lang_year	2	release_year	A	1			YES	BTREE
def	sakila	film_temp	0	sakila	PRIMARY	1	film_id	A	1000				BTREE

Индексы

Получим фильмы по двойному условию с составным индексом:

```
EXPLAIN
SELECT *
FROM film_temp
WHERE language_id = 1 AND release_year = 2006;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film_temp		ref	lang_year	lang_year	10	const,const	500	100.0	

Планировщик использует составной индекс **lang_year**:

```
EXPLAIN ANALYZE
SELECT *
FROM film_temp
WHERE language_id = 1 AND release_year = 2006;
-> Index lookup on film_temp using lang_year (language_id=1, release_year=2006)
(cost=80.53 rows=500) (actual time=0.038..1.409 rows=500 loops=1)
```



Типы индексов

Индексы

В MySQL индексы можно разделить на следующие типы:

- **B-TREE** — PRIMARY KEY, UNIQUE, INDEX и FULLTEXT,
- **R-TREE** — пространственные типы данных,
- **INVERTED** — в механизме хранения InnoDB для FULLTEXT,
- **HASH** — только в механизме хранения Memory.

B-TREE

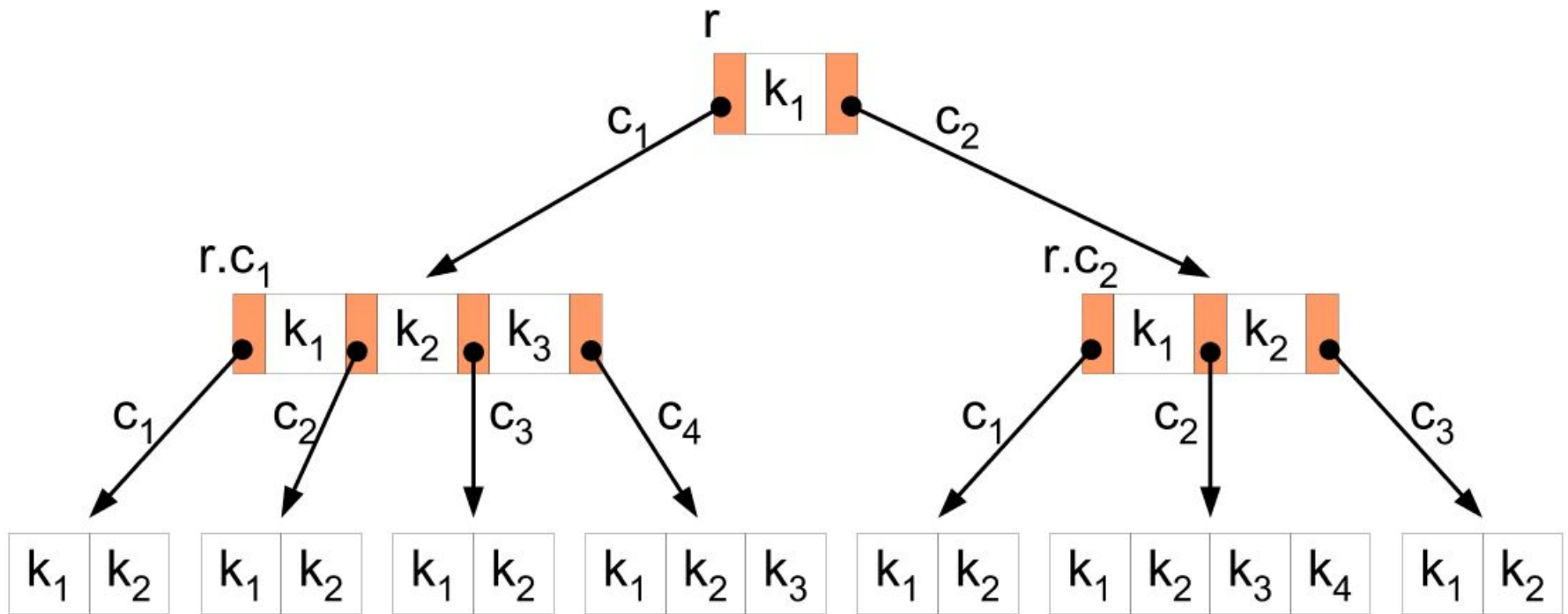
B-TREE — древовидная структура данных, популярная для использования в индексах БД.

Структура всегда отсортирована, что обеспечивает быстрый поиск точных совпадений (оператор равенства) и диапазонов (например, больше, меньше и операторов **BETWEEN**).

Этот тип индекса доступен для большинства механизмов хранения, поскольку узлы **B-TREE** могут иметь много дочерних элементов.

B-TREE не то же самое, что двоичное дерево, которое ограничено двумя дочерними элементами на узел.

B-TREE



R-TREE

R-дерево (англ. R-trees) — древовидная структура данных (дерево), предложенная в 1984 году Антонином Гуттманом.

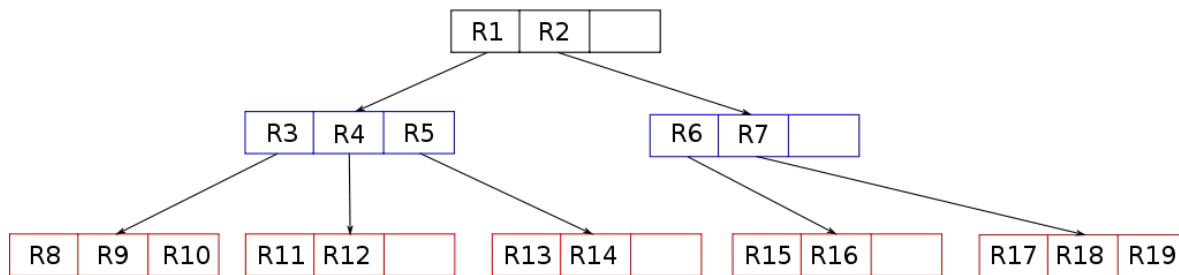
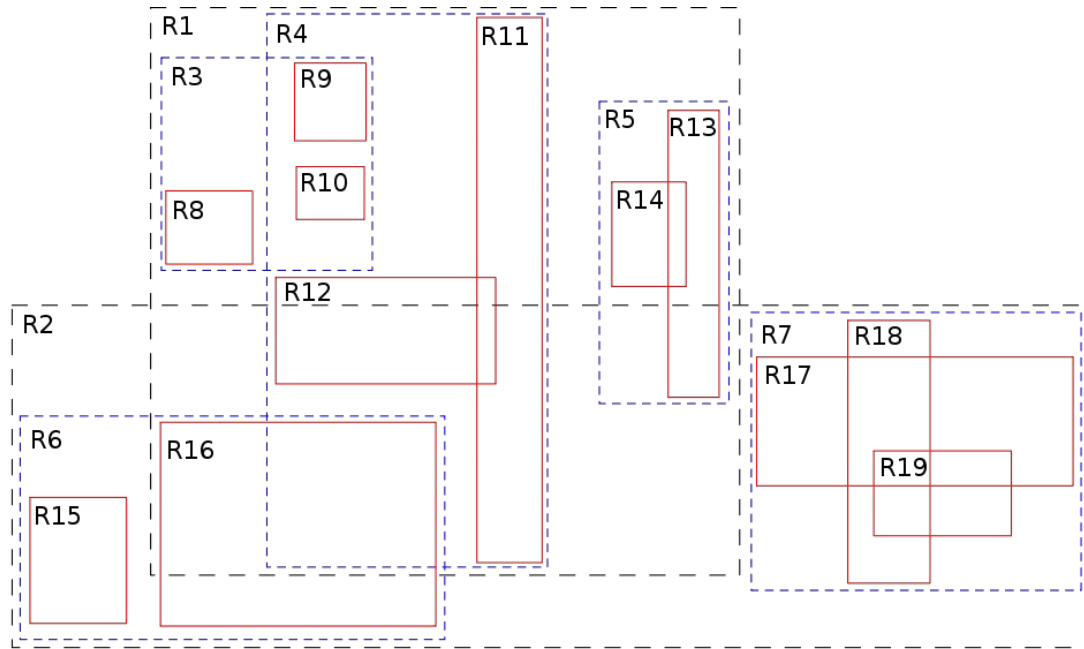
Она подобна B-дереву, но используется для организации доступа к пространственным данным, то есть для индексации многомерной информации, такой, например, как географические данные с двумерными координатами (широтой и долготой).



R-TREE

Эта структура данных разбивает многомерное пространство на множество иерархически вложенных и, возможно, пересекающихся, прямоугольников (для двумерного пространства). В случае трехмерного или многомерного пространства это будут прямоугольные параллелепипеды (кубоиды) или параллелотопы.

R-TREE



INVERTED

Инвертированный индекс – это полнотекстовый индекс, структура данных, в которой для каждого слова коллекции документов в соответствующем списке перечислены все документы в коллекции, в которых оно встретилось.

Возьмем таблицу **film** и столбцы **film_id** и **description**, выведем первые три строки:

```
SELECT film_id, description
FROM film
LIMIT 3;
```

film_id	description
1	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The Canadian Rockies
2	A Astounding Epistle of a Database Administrator And a Explorer who must Find a Car in Ancient China
3	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack in A Baloon Factory

INVERTED

Инвертированный индекс будет выглядеть следующим образом:

```
'A':          {1,2,3}
'Epic':       {1}
'And':        {1,2,3}
'Astounding': {2,3}
'Database':   {2}
...
```

Прямой бы индекс выглядел следующим образом:

```
1: {'A', 'Epic', 'And'}
2: {'A', 'And', 'Astounding', 'Database'}
3: {'A', 'And', 'Astounding'}
...
```

HASH

HASH-индексы были предложены Артуром Фуллером, и предполагают хранение не самих значений, а их хэшей, благодаря чему уменьшается размер и увеличивается скорость обработки индексов из больших полей. Таким образом, при запросах с использованием **HASH**-индексов, сравниваться будут не искомое со значением поля, а хэш от искомого значения с хэшами полей.



HASH

Из-за нелинейности хэш-функций данный индекс нельзя сортировать по значению, что приводит к невозможности использования в сравнениях больше/меньше и «is null». Кроме того, так как хэши не уникальны, то для совпадающих хэшей применяются методы разрешения коллизий.

ADAPTIVE HASH

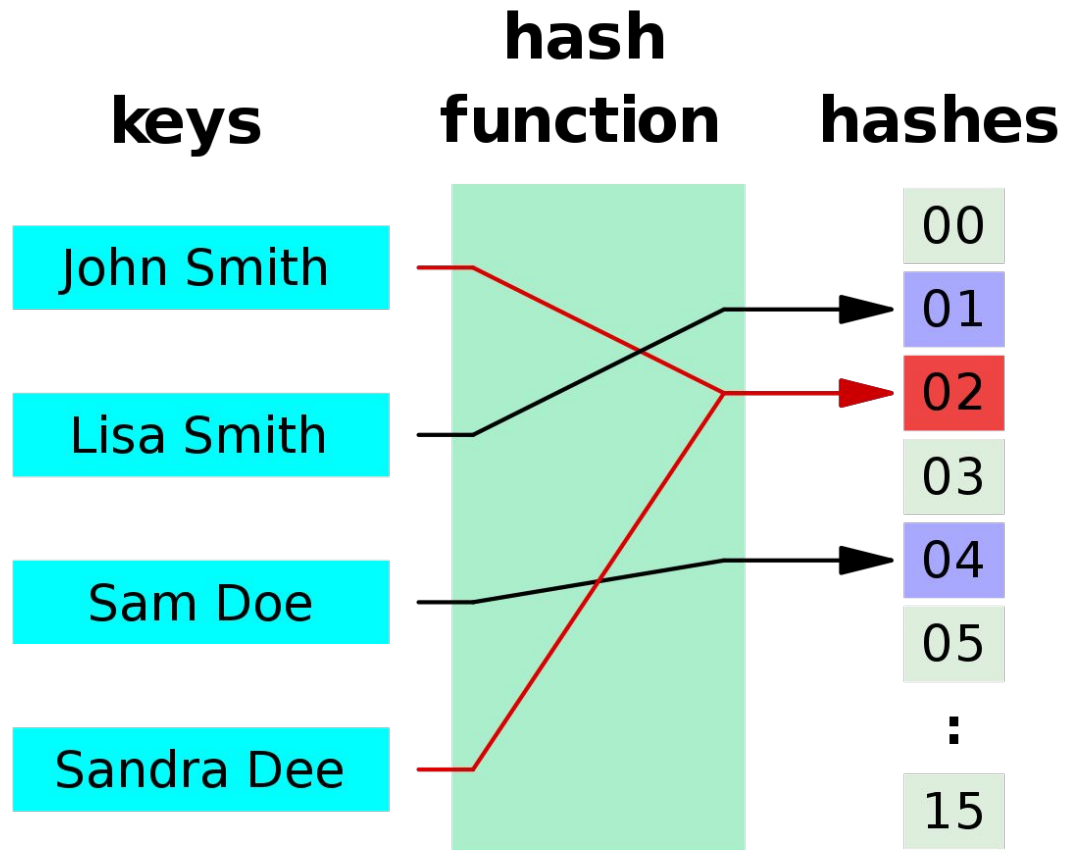
ADAPTIVE HASH используется для оптимизации **InnoDB** таблиц, которые могут ускорить поиски с использованием строгого равенства и **IN** операторами, путем построения **HASH**-индекса в памяти. **MySQL** отслеживает поиск по индексу для **InnoDB** таблиц, и если запросы могут выиграть от **HASH**-индекса, он автоматически строит индекс для часто используемых страниц индекса.

ADAPTIVE HASH

В некотором смысле **ADAPTIVE HASH** настраивает **MySQL** во время выполнения, чтобы использовать преимущества большой основной памяти, приближаясь к архитектуре баз данных с основной памятью.

ADAPTIVE HASH всегда строится на основе существующего индекса **B-TREE** в таблице.

HASH



Уникальные индексы

Индексы разделяются на **обычные** и **уникальные**.

Когда происходит поиск по обычному индексу, то есть по не уникальным значениям, то после первого нахождения соответствия поиск будет продолжен.

В случае с уникальным индексом, после нахождения искомого значения поиск будет остановлен.

Уникальные ключи работают так же, как и первичные ключи, только их может быть любое количество. Создаются через запросы:

```
ALTER TABLE `имя_таблицы` ADD UNIQUE INDEX `имя_индекса` ...;  
CREATE UNIQUE INDEX `имя_индекса` ...;
```

Кластерные индексы

Обычные индексы являются не кластерными, и сам индекс хранит только ссылки на записи таблицы. Когда происходит работа с индексом, определяется только список первичных ключей записей, подходящих под запрос. После этого происходит еще один запрос — для получения данных каждой записи из этого списка.

Кластерные индексы сохраняют данные записей целиком, а не ссылки на них. При работе с таким индексом не требуется дополнительной операции чтения данных. Первичные ключи таблиц InnoDB являются кластерными и выборки по ним происходят эффективно.



Стоимость индексов

Стоимость индексов

При использовании индексов происходят дополнительные операции записи на жесткий диск. Таким образом при каждом обновлении или добавлении данных в таблицу, происходит также запись и обновление данных в индексе.

Если в операциях **SELECT** индексы ускоряют работу, то в операциях **INSERT** и **UPDATE** время увеличивается, как и занимаемое место на жестком диске.

Удалим индексы во временной таблице **film_temp**:

```
ALTER TABLE film_temp DROP PRIMARY KEY;  
DROP INDEX lang_year ON film_temp;
```

Стоимость индексов

Посмотрим на размер таблицы без индексов:

```
SELECT table_name, data_length, index_length
FROM INFORMATION_SCHEMA.TABLES
WHERE table_name = "film_temp";
```

TABLE_NAME	DATA_LENGTH	INDEX_LENGTH
film_temp	180224	0

Добавим индекс **PRIMARY KEY** и составной индекс **lang_year** и проверим размер:

```
ALTER TABLE film_temp ADD PRIMARY KEY (film_id);
CREATE INDEX lang_year ON film_temp(language_id, release_year);
SELECT table_name, data_length, index_length
FROM INFORMATION_SCHEMA.TABLES
WHERE table_name = "film_temp";
```

TABLE_NAME	DATA_LENGTH	INDEX_LENGTH
film_temp	180224	49152



Итоги

Итоги

В данной лекции мы:

- Научились использовать планировщик запросов **EXPLAIN**;
- Разобрали, какие типы индексов существуют в **MySQL**;
- Посмотрели на механику работы индексов и их стоимость.





Домашнее задание



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

Роман Гордиенко