Оптимизация MySQL

Отдел R&D

Маркетинговая группа Текарт

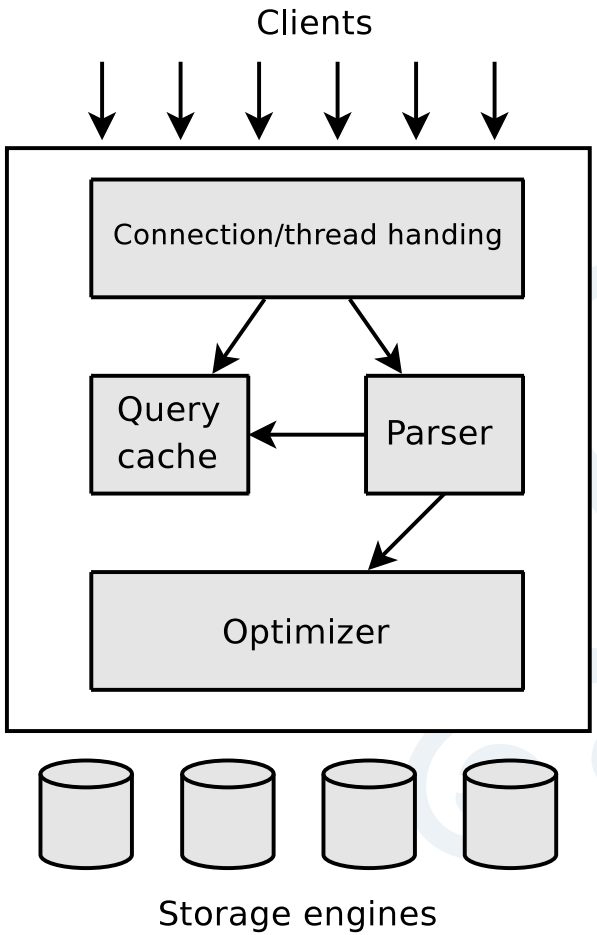
23 июля 2009 г.

О чем речь

1. Введение
2. Механизмы хранения
3. Типы данных
4. Индексы
5. Нормализация и денормализация
6. Merge & Partitioning
7. Оптимизация запросов
8. Кеширование запросов
9. Хранимые процедуры и функции
10. Представления

О чем речь

Архитектура MySQL



Университет Текарт

Lock & Transactions

Lock

- Table Lock - MyISAM
- Row Lock - InnoDB, Falcon

Transaction ACID

- Atomicity (атомарность)
- Consistency (непротиворечивость)
- Isolation (изоляция)
- Durability (долговечность)

Если транзакции не нужны - отключите их, т.к. это дополнительная нагрузка на CPU, память и т.д.

Некоторые термины

Dirty read возможность читать незакоммиченные данные;

Nonrepeatable read выборка одного и того же значения в разные моменты времени выполнения транзакции может выдать разные результаты;

Phantom read в результат выборки интервала строк могут попасть новые строки из другой транзакции;

Locking read блокировка каждой читаемой строки.

ANSI SQL isolation levels

Isolation level	Dirty read possible	Nonrepeatable reads possible	Phantom read possible	Locking reads
READ UNCOMMITED	ДА	ДА	ДА	НЕТ
READ COMMITED	НЕТ	ДА	ДА	НЕТ
REPEATABLE READ	НЕТ	НЕТ	ДА	НЕТ
SERIALIZABEL	НЕТ	НЕТ	НЕТ	ДА

Транзакции

- Поддержка: InnoDB, Falcon, ...
- `REPEATABLE READ` по умолчанию (в отличие от большинства других СУБД),

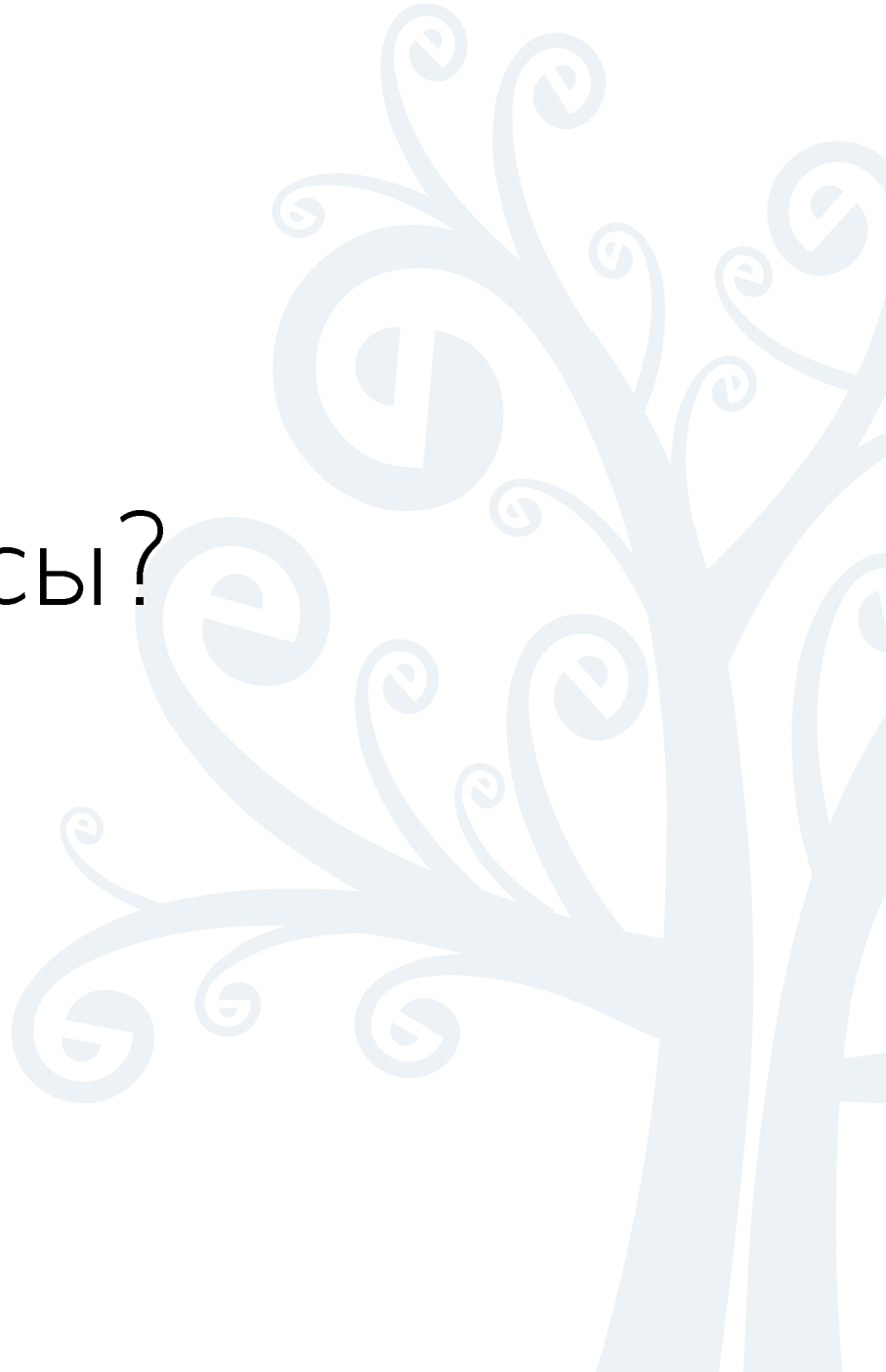
  `SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED`
- `AUTOCOMMIT` по умолчанию, `SET AUTOCOMMIT = 0`
- Для InnoDB можно явно указывать locking

  `SELECT ... LOCK IN SHARE MODE...`

MVCC: Multiversion Concurrency Control

- каждая транзакция работает со "снимком" данных в момент начала транзакции;
- пишущие транзакции не блокируют читающих, и читающие транзакции не блокируют пишущих
- реализовано в InnoDB и Falcon, но везде по разному.

Вопросы?



О чем речь

Особенности MyISAM

- table locking, shared read locks, exlusive write locks;
- сравнительно быстрый `insert`;
- manual repair (InnoDB – гораздо лучше);
- нет транзакций;
- индексирует первые 500 символов в `BLOB` и `TEXT`, поддержка full-text indexes;
- delayed key writes – возможность не сразу писать индексы на диск, снижает нагрузку, но можно потерять индекс при сбое;
- myisampack: возможность сжатия таблиц;
- MyISAM merge engine – объединение нескольких таблиц в одну

Особенности InnoDB

- транзакции;
- поддержка внешних ключей;
- crash recovery – высокая степень надежности, часто именно из-за этого выбирают InnoDB вместо MyISAM ;
- tablespace – хранение данных на разных физических носителях;
- параллелизм – MVCC, REPEATABLE READ isolation level, нет phantom reads;
- clustered index – быстрый поиск по первичному ключу;
- вторичные индексы содержат в себе первичные, что увеличивает их размер;
- разрабатывалось для слабых компьютеров, на смену должен прийти Falcon.

Другие механизмы хранения

Maria	должен заменить MyISAM (6.x), alfa;
Falcon	должен заменить InnoDB (6.x), хорошие результаты уже сейчас;
Memory	хранит данные в памяти, часто используется для анализа данных (быстрая обработка);
Archive	используется соответственно для архивных данных, хорошо подходит для логирования;
CSV	удобно для хранения данных в CSV формате.
...	

Если таблица содержит какие-либо специфичные данные или используется нестандартно, стоит поискать для этих цели storage engine

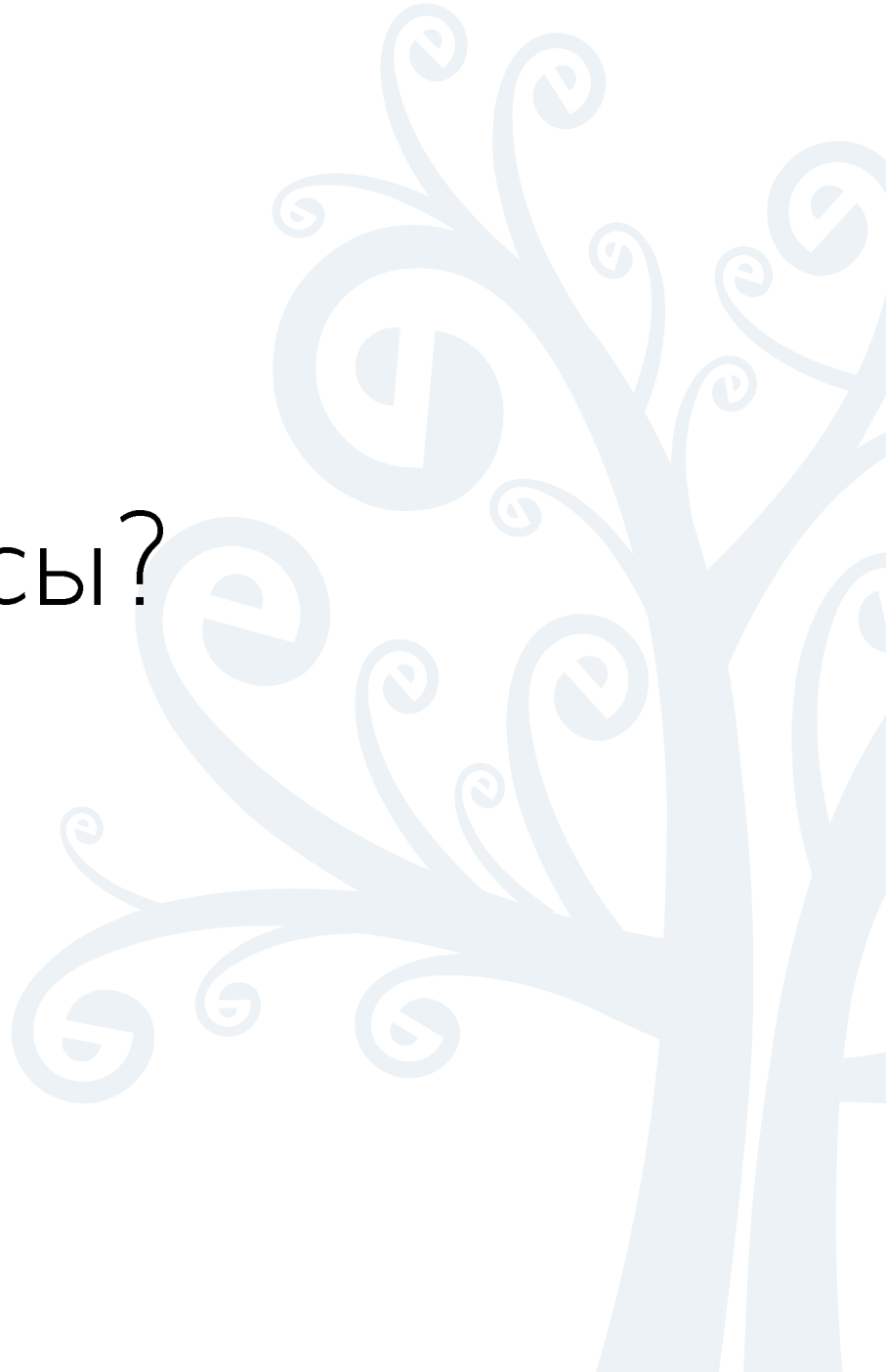
Характеристики различных механизмов хранения

Storage engine	MySQL version	Transactions	Lock granularity	Key applications	Counter-Indication
MyISAM	All	No	Table with concurrent inserts	SELECT, INSERT, bulk loading	Mixed read/write workload
MyISAM Merge	All	No	Table with concurrent inserts	Segmented archiving, data warehousing	Many global lookups
Memory (HEAP)	All	No	Table	ntermediate calculations, static lookup data	Large datasets, persistent storage
InnoDB	All	Yes	Row-level with MVCC	Transactional processing	None
Falcon	6.0	Yes	Row-level with MVCC	Transactional processing	None
Archive	4.1	Yes	Row-level with MVCC	Logging, aggregate analysis	Random access needs, updates, deletes
CSV	4.1	No	Table	Logging, bulk loading of external data	Random access needs, indexing
Blackhole	4.1	Yes	Row-level with MVCC	Logged or replicated archiving	Any but the intended use
Federated	5.0	N/A	N/A	Distributed data sources	Any but the intended use
NDB Cluster	5.0	Yes	Row-level	High availability	Most typical uses
PBXT	5.0	Yes	Row-level with MVCC	Transactional processing, logging	Need for clustered indexes
solidDB	5.0	Yes	Row-level with MVCC	Transactional processing	None
Maria (planned)	6.x	Yes	Row-level with MVCC	MyISAM replacement	None

Общие рекомендации по выбору механизма хранения

- не требуются транзакции, SELECT, INSERT – MyISAM;
- транзакции, сложные запросы, UPDATE – InnoDB;
- InnoDB лучше подходит, если требуется высокая надежность
- специфические задачи – специфические механизмы хранения.

Вопросы?



О чем речь

Выбор типов данных

Общие рекомендации:

- по возможности выбирать типы данных, занимающие меньше места;
- использовать наиболее простые типы (например IP-адреса хранить в виде числа, а не строки);
- NOT NULL везде, где только можно.

Университет Текарт

Выбор типов данных: числа

- по возможности использовать `INT` (`TINYINT` ...);
- по возможно использовать `UNSIGNED`;
- для больших значений использовать `DECIMAL`;
- По возможности использовать `FLOAT` для чисел с плавающей точкой (вместо `DOUBLE` и `DECIMAL`);
- `DECIMAL` хорошо подходит для финансовых данных;
- `INT(1)` и `INT(20)` – одинаковое количество памяти :-).

Выбор типов данных: строки

- VARCHAR – дополнительно один или два байта для хранения длины, CHAR(1) занимает меньше места, чем VARCHAR(1);
- CHAR – хранение коротких строк или если точна известна длина строки, обновление, сортировка и т.д. происходят быстрее;
- VARCHAR – меньше места, но операции обновления, построения индекса и т.д. могут занимать больше времени.
- некоторые storage engines не поддерживают данные с переменной длиной $\Rightarrow$ каждый раз вычисляют максимальное значение и переформатируют данные $\Rightarrow$ увеличение нагрузки;
- некоторые наоборот – CHAR в виде VARCHAR (Falcon).

Выбор типов данных: BLOB и TEXT

Созданы для хранения больших объемов данных, но:

- индексируется только max_sort_length символов
- каждая storage engine хранит эти объекты по своему, в InnoDB – отдельное хранилище;
- temporary table и memory engine не поддерживают этот тип данных $\Rightarrow$ вместо таблицы в памяти создается таблица на диске, что очень влияет на производительность таких запросов как JOIN.

Выбор типов данных: ENUM

Можно использовать вместо CHAR и VARCHAR т.к. занимает меньше места, быстрее сортируется и т.д., но:

- поле на самом деле хранит числа, т.е. `SELECT e + 0 FROM enum_test;` вернет число;
- значения сортируются по числам, а не по строкам:

```
SELECT e FROM enum_test
ORDER BY FIELD(e, 'apple', 'dog', 'fish');
```

- JOIN с использованием полей ENUM - ENUM происходит быстро, а вот ENUM - VARCHAR медленно, в этом случае лучше использовать VARCHAR - VARCHAR;
- для добавления нового значения требуется `ALTER TABLE`.

Выбор типов данных: DATETIME vs TIMESTAMP

По возможности используйте TIMESTAMP:

- DATETIME включает больший диапазон дат: от 1001 до 9999 года, с точностью до секунды;
- TIMESTAMP занимает в два раза меньше места;
- TIMESTAMP зависит от зоны, DATETIME – нет;
- MySQL по умолчанию подставляет текущую дату в TIMESTAMP, но только для первого поля;
- TIMESTAMP по умолчанию NOT NULL, если несколько полей TIMESTAMP – не забываем писать `DEFAULT NULL`;

Выбор типов данных: bit-packed

- BIT: хранение битовых данных, мало место и быстрая обработка;
- внимание: значение b'00111001'(57) будет возвращено как '9', (57 в ASCII таблице) $\Rightarrow$ для получения числового значения используйте field+0;

Вместо INT можно использовать SET:

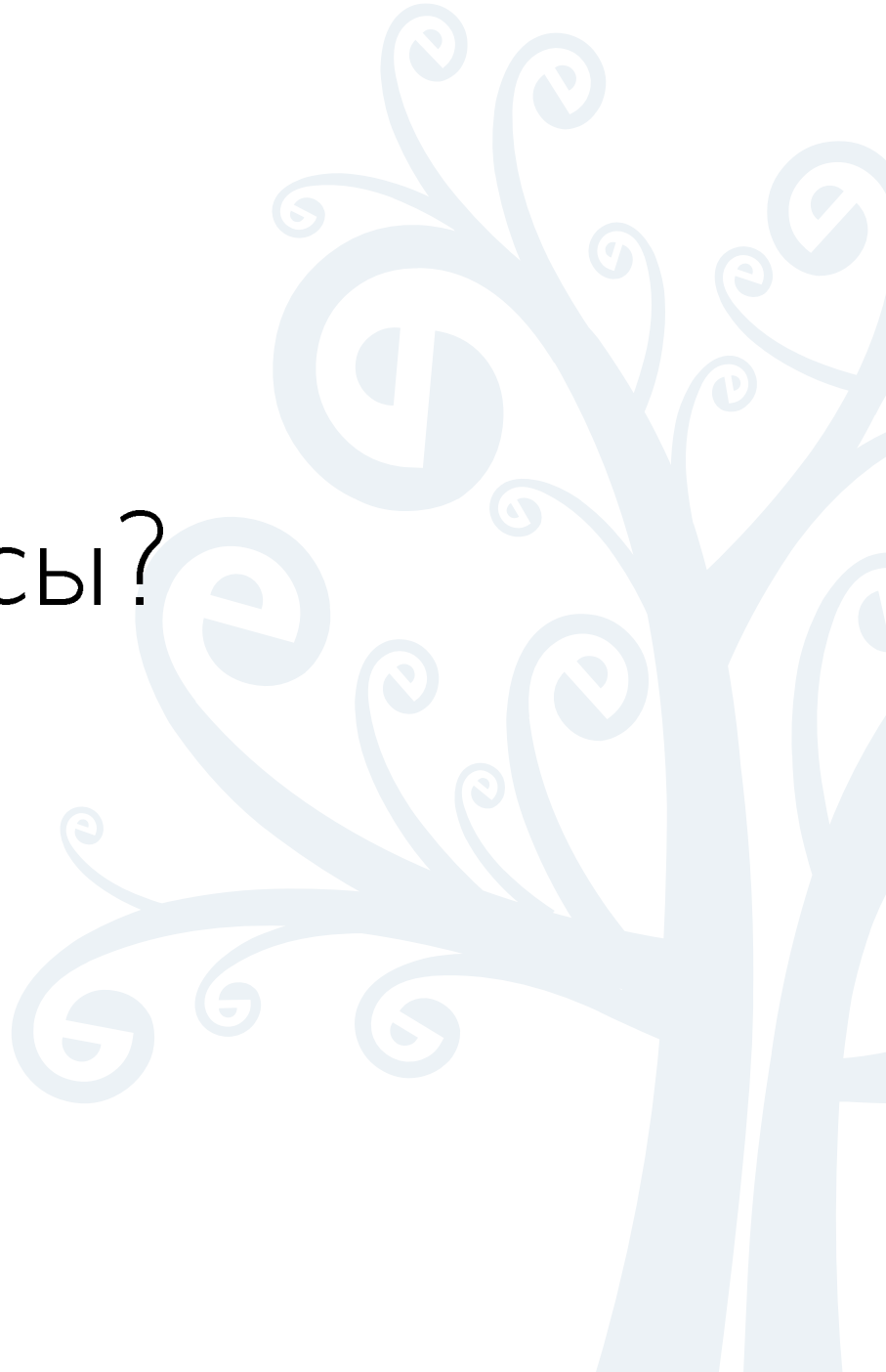
```
CREATE TABLE acl
  (perms SET('CAN_READ', 'CAN_WRITE', 'CAN_DELETE') NOT NULL);

INSERT INTO acl(perms) VALUES ('CAN_READ,CAN_DELETE');
SELECT perms FROM acl WHERE FIND_IN_SET('CAN_READ', perms);
```

Выбор типа для идентификатора

Выбираем самый простой тип:

- INT + AUTO_INCREMENT – лучший выбор (особенно для InnoDB);
- ENUM или SET – в общем случае плохой выбор, но подходит для статических справочников;
- Строковые типы – много места, медленно обрабатываются. Не рекомендуется использовать в MyISAM, т.к. по умолчанию MyISAM жмет ключи $\Rightarrow$ дополнительная нагрузка при использовании строк;
- Не желательно использовать случайные строки (MD5, SHA1) – замедляет выборку и вставку.

Вопросы?



О чем речь

Типы индексов

Типа зависит от механизма хранения, но чаще всего – b-tree.

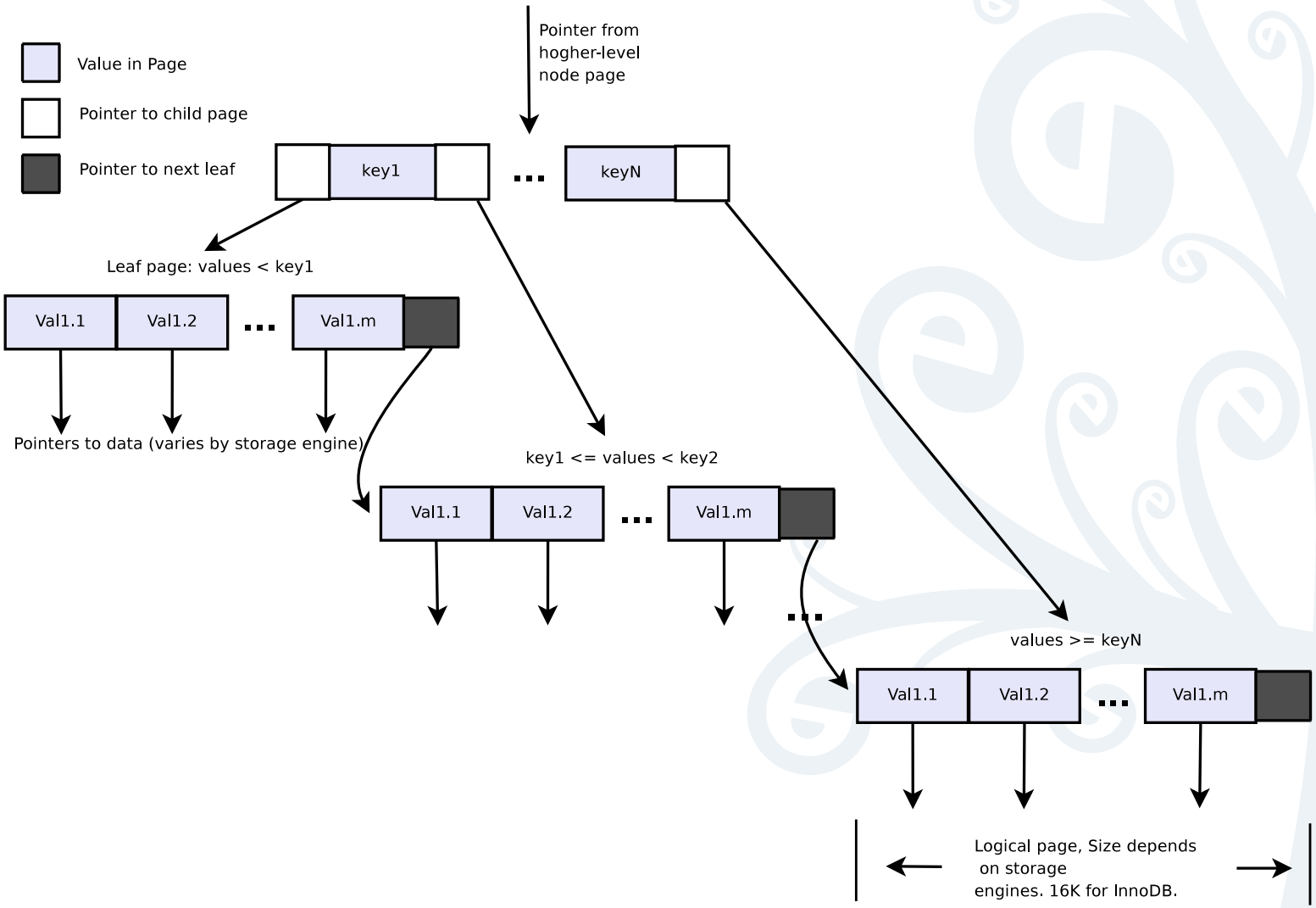
B-Tree основной тип, подходит для сравнения по полному значению, для выражений с промежутками (BETWEEN, <,> ...), является префиксным, что накладывает ряд ограничений;

Hash использует хэш-функцию для формирования, хорошо подходит для сравнения по полному значению индекса, не префиксный;

R-Tree аналог B-Tree, используется MyISAM для полнотекстового поиска;

Некоторые механизмы хранения используют собственные индексы, каждый механизм хранит индексы по-своему.

B-Tree



Университет Текарт

Где могут использоваться B-Tree-индексы

Поиск:

- по полному значению индекса;
- по левостороннему префиксу индекса, например по первой колонке в индексе;
- по началу колонки в индексе, т.е. найти всех сотрудников имя которых начинается на 'A';
- по интервалу значений, но только по первой колонке;
- при фиксированном значении первой колонки (или нескольких колонок) и при интервале на последующую;

Могут использоваться в выражениях, которые оперируют только индексами: covering indexes.

Ограничения B-Tree индексов

- нельзя искать значения оканчивающиеся на что-то. т.е. B-Tree индекс не поможет найти всех сотрудников, имена которых заканчиваются на 'Я';
- нельзя пропустить колонку в индексе;
- нет возможности оптимизировать выражения типа:

```
WHERE last_name="Smith" AND
      first_name LIKE 'J%' AND
      date_of_birth = '1976-12-23'
```

значения `date_of_birth` не будет использоваться в индексном поиске.

Hash-индексы

Очень быстрый поиск по полному значение индекса, но есть ограничения:

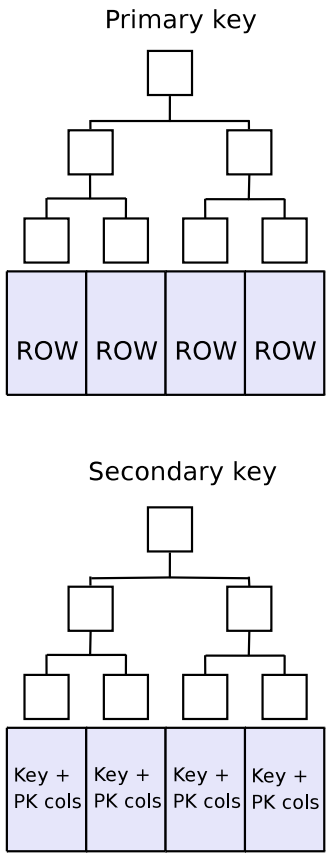
- нельзя использовать для covering indexes;
- нельзя использовать для сортировки;
- не могут использоваться в выражениях <, >, только в выражениях =, IN(), <=>;
- не эффективен при частых коллизиях.

Стратегии использования индексов

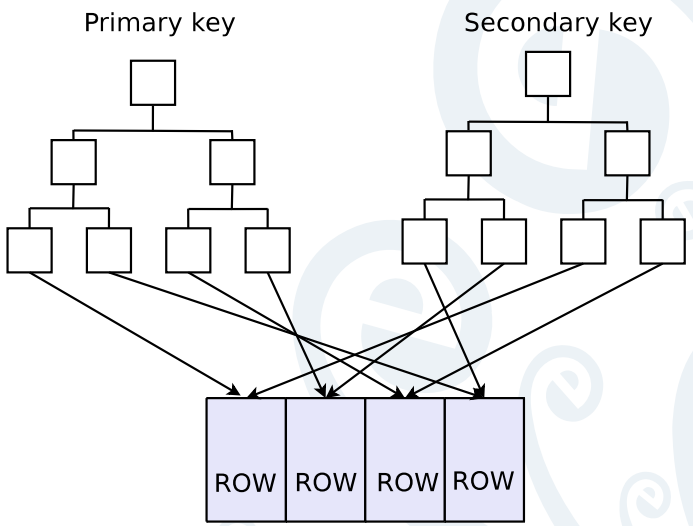
- Изолирование колонки индекса, колонка не должна участвовать в выражении или быть внутри функции.
- Индекс слишком большой – префиксные индексы по первым символам колонки. При этом стараемся сохранить максимальную селективность (отношение различных значений индекса к количеству строк в таблице).
- Избегаем слишком больших индексов и дублирования индексов.
- Индексы строятся под конкретное выражение и часто бывает выгодно дублировать индекс. Дублирование меньше сказывается на производительности в InnoDB, чем в MyISAM;
- Индексы тесно связаны с блокировками, особенно в InnoDB.
- Covering indexes – способ избежать чрезмерного блокирования строк.

Кластеризованные индексы

Реализованы только в solidDB и InnoDB



InnoDB (clustered) table layout



MyISAM (non-clustered) table layout

Когда полезны кластеризованные индексы

- можно хранить связанные данные близко друг к другу и считывать их за одну операцию чтения;
- при использовании связи по первичному ключу данные извлекаются быстро. т.к. храняться вместе с ключом
- покрывающие индексы могут дополнительно использовать первичный ключ;
- для данных в памяти роста производительности не будет;
- быстрая вставка строк в порядке сортировки первичного ключа, в других случаях медленно;
- при обновлении первичного ключа нужны дополнительные операции чтобы переместить строку;
- вторичные ключи занимают больше места, т.к. хранят значения первичных;
- при поиске по вторичному ключу дополнительно производиться просмотр по первичному ключу.

Покрывающие индексы

Индексы, которые покрывают все необходимые данные. В этом случае MySQL берет значения непосредственно из индексов.

- значение индекса занимает много меньше места чем вся стока из таблицы;
- доступ к данным индекса производить быстрее чем к строке таблицы;
- зачастую индексы кешируются лучше чем данные;
- в InnoDB вторичные ключи содержат значения первичных, что дает дополнительный покрывающий индекс.

Покрывающие индексы: примеры

```
SELECT store_id, film_id FROM sakila.inventory; —при
индексе на (store_id, film_id)


SELECT * FROM products WHERE actor='SEAN CARREY'
AND title like '%APOLLO%' ->
SELECT *
    FROM products
      JOIN (
         SELECT prod_id
         FROM products
       WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
      ) AS t1 ON (t1.prod_id=products.prod_id)
```

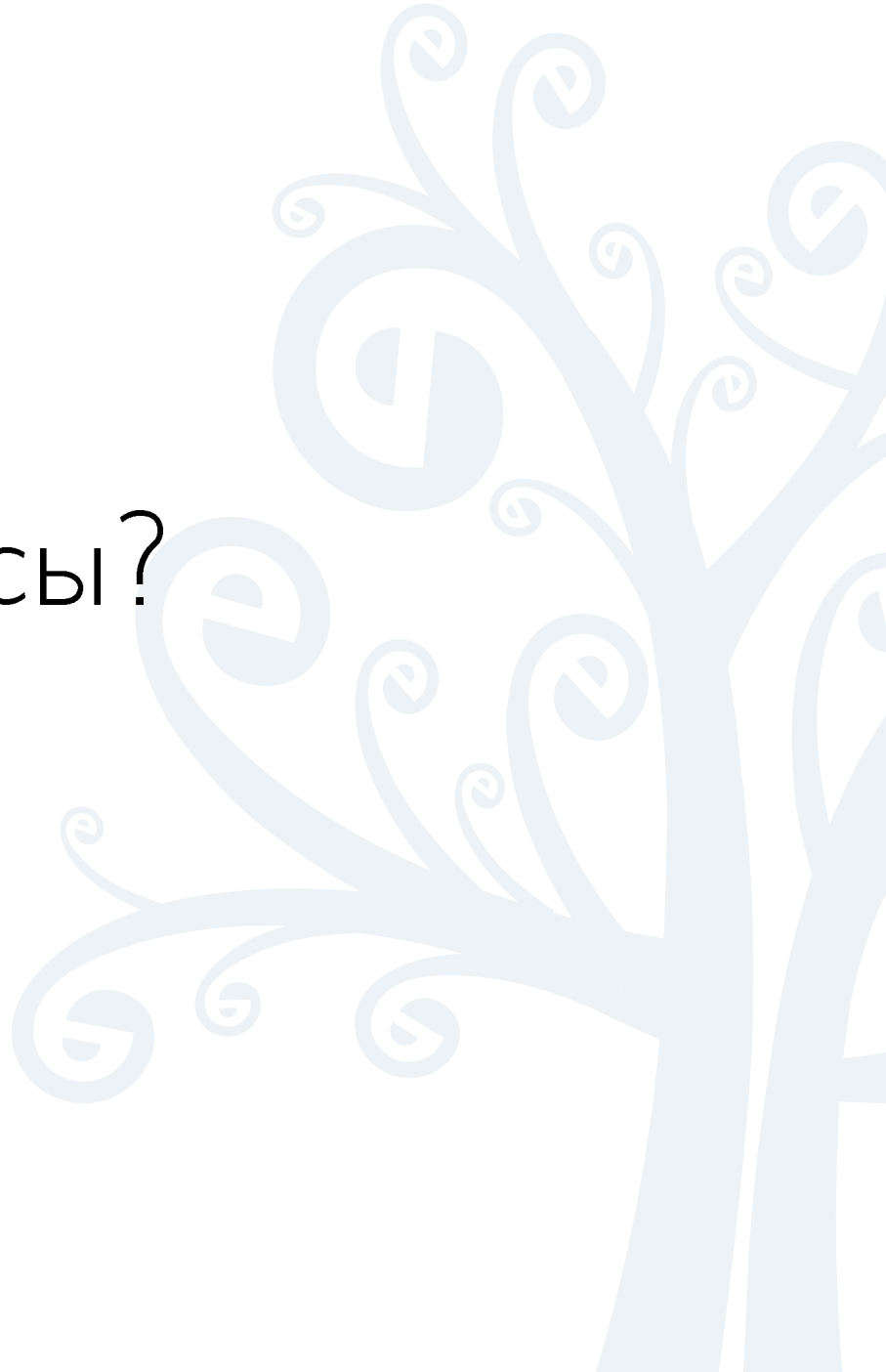
Использование индексов для сортировки

Основано на свойстве левосторонней префиксности индексов.

Примеры, когда индексы не используются:

```
CREATE TABLE rental ( ...
UNIQUE KEY rental_date (rental_date,inventory_id,customer_id) ...);

WHERE rental_date = '2005-05-25' ORDER BY
  inventory_id DESC, customer_id ASC;

WHERE rental_date = '2005-05-25' ORDER BY
  inventory_id, staff_id;

WHERE rental_date = '2005-05-25' ORDER BY
  customer_id;

WHERE rental_date > '2005-05-25' ORDER BY
  inventory_id, customer_id;

WHERE rental_date = '2005-05-25' AND inventory_id IN(1,2) ORDER BY
customer_id;
```

Вопросы?



О чем речь

Университет Текарт

Нормализация и денормализация в MySQL

Нормализированная схема:

- быстрее обновляется;
- меньше дублирующихся данных;
- нет необходимости писать `DISTINCT` и `GROUP BY` запросы.

Денормализированная схема:

- позволяет избежать `JOIN`, что в случае MySQL дает значительный выигрыш;
- используйте специальные таблицы: cache, summary, counter.

Таблицы cache, summary, counter

Cache Создается для оптимизации сложных выборок. Т.е. создается специальная таблица, содержащая только те данные которые необходимо из другой таблицы или нескольких таблиц + создаются необходимые индексы.

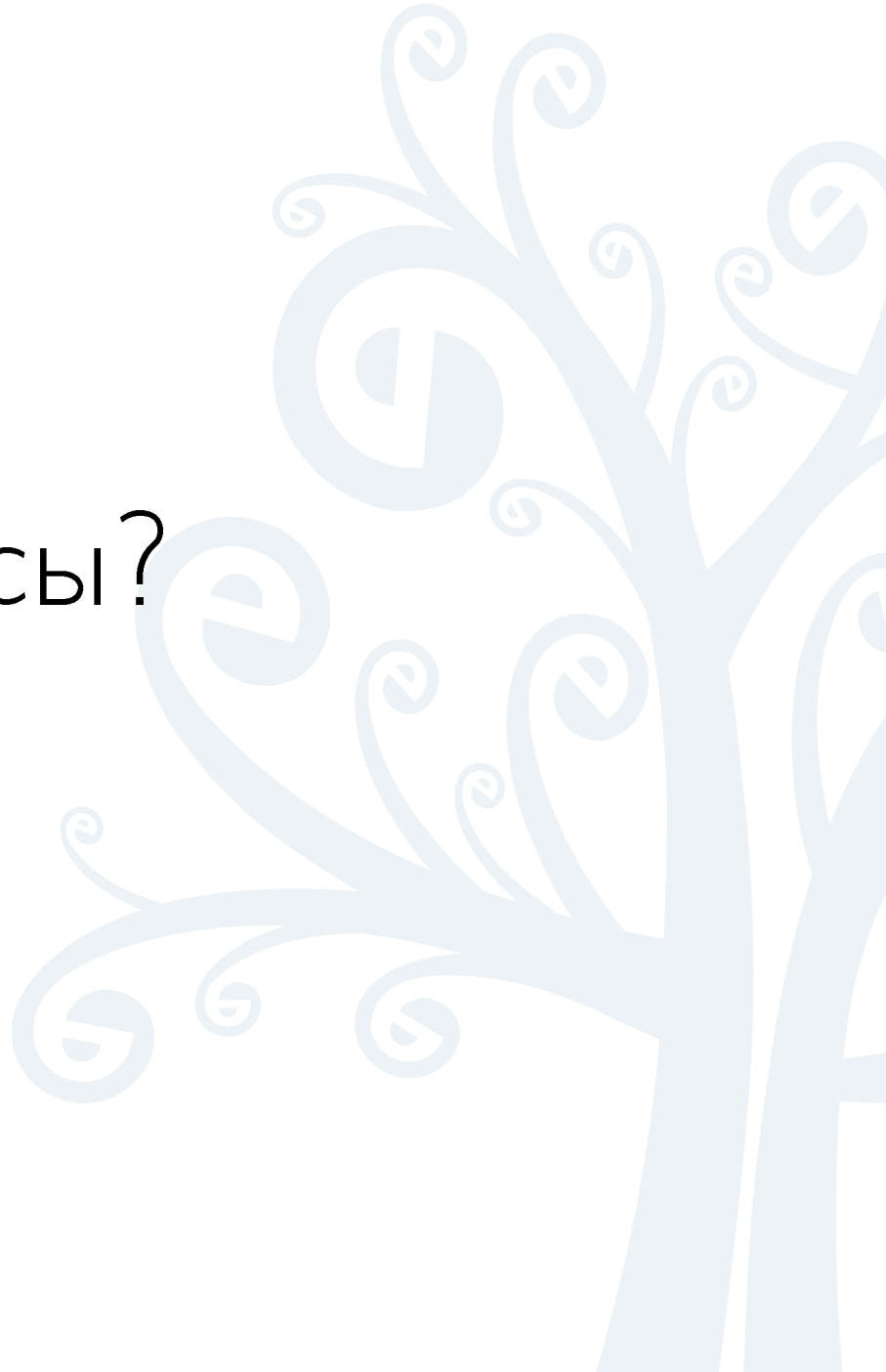
Summary Специальные таблицы которые содержат агрегированные данные от запросов `GROUP BY`. Например таблица хранящая информацию о количестве сообщений каждый час.

Counter Специальные таблицы хранящие количество скачек файлов, просмотров страницы, кликов и т.д.

Примеры cache, summary, counter таблиц

```
CREATE TABLE msg_per_hr (
  hr DATETIME NOT NULL,
  cnt INT UNSIGNED NOT NULL,
  PRIMARY KEY(hr)
);

SELECT SUM(cnt) FROM msg_per_hr
WHERE hr BETWEEN
   CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 23 HOUR
   AND CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 1 HOUR;

CREATE TABLE hit_counter (
  slot tinyint unsigned not null primary key,
  cnt int unsigned not null
) ENGINE=InnoDB;

INSERT INTO daily_hit_counter(day, slot, cnt)
  VALUES(CURRENT_DATE, RAND( ) * 100, 1)
  ON DUPLICATE KEY UPDATE cnt = cnt + 1;
```

Вопросы?



О чем речь

Университет Текарт

Merge Table: особенности

Объединяет несколько таблиц в одну, как VIEWS + UNION

- всегда держит открытыми несколько дескрипторов файла;
- не проверяет на соответствие таблицы входящие в объединение, просто выдаст ошибку при обращении;
- чтение таблиц – в порядке их объявления при создании merge table;
- выбор одной строки по ключу происходит медленней;
- поиск интервала значений происходит быстрей;
- создание, удаление и модификация merge tables – быстрая операция.

Merge Tables: пример

```
CREATE TABLE t1(a INT NOT NULL PRIMARY KEY) ENGINE=MyISAM;
CREATE TABLE t2(a INT NOT NULL PRIMARY KEY) ENGINE=MyISAM;
INSERT INTO t1(a) VALUES(1),(2);
INSERT INTO t2(a) VALUES(1),(2);
CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
  ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
SELECT a FROM mrg;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

Merge Table – некоторые стратегии использования

- для доступа к данным, имеющим активную и неактивную часть, например, логирование: каждый день создавать таблицу, в которую производить вставку, и добавлять её в merge table;
- разбивать большие таблицы на маленькие: их проще содержать, востанавливать и т.д.
- удобно для удаления старых данных: быстрее удалить целиком маленькую таблицу;
- Использовать вместо VIEW + UNION: работает быстрее т.к. не создает временных таблиц. Например можно объединять вчерашние данные с предыдущими, для предоставления недельных отчетов;
- можно сжимать “старые” таблицы.

Partitions: приемущества и ограничения

- можно указать, какие именно строки входят в отдельную часть: например, при разделении по дате, выражения с выборкой по дате будут обращаться только к одной части;
- частями легче управлять, чем целыми данными;
- части можно разместить на разных физических носителях
- нельзя обратится к частям по отдельности, как в случае merge table;
- для разбиения на части используется ограниченное множество partitions functions;
- грубо partitions можно воспринимать как особый вид индексов;
- каждый уникальный ключ должен содержать ссылку на partitin function, что увеличивает его размер.
- внешние ключи не работают;
- не все хранимые механизмы поддерживают.

Partitions: примеры использования

```
CREATE TABLE sales_by_day (
   day DATE NOT NULL,
   product INT NOT NULL,
   sales DECIMAL(10, 2) NOT NULL,
   returns DECIMAL(10, 2) NOT NULL,
   PRIMARY KEY(day, product)
) ENGINE=InnoDB;

ALTER TABLE sales_by_day
  PARTITION BY RANGE(YEAR(day)) (
    PARTITION p_2006 VALUES LESS THAN (2007),
    PARTITION p_2007 VALUES LESS THAN (2008),
    PARTITION p_2008 VALUES LESS THAN (2009),
    PARTITION p_catchall VALUES LESS THAN MAXVALUE );

   ALTER TABLE mydb.very_big_table
   PARTITION BY KEY(<primary key columns>) (
      PARTITION p0 DATA DIRECTORY='/data/mydb/big_table_p0/',
      PARTITION p1 DATA DIRECTORY='/data/mydb/big_table_p1/');
```

Partitions – примеры оптимизации запросов

EXPLAIN PARTITIONS – выводит используемые части

```
EXPLAIN PARTITIONS
  SELECT * FROM sales_by_day WHERE day > '2007-01-01'
...
partitions: p_2007,p_2008
```

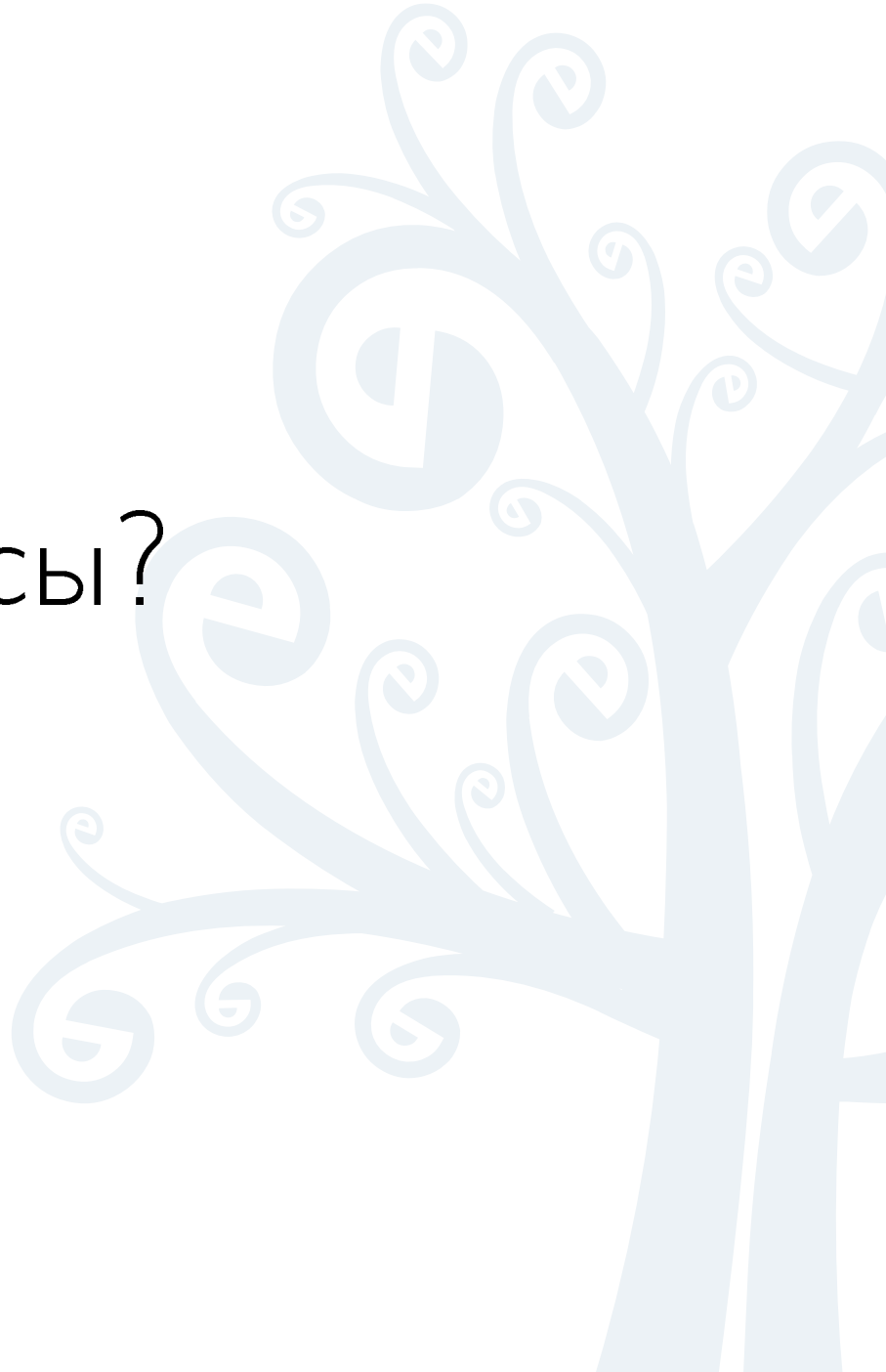
MySQL не всегда понимает когда применить partitions:

```
EXPLAIN PARTITIONS SELECT *
  FROM sales_by_day WHERE YEAR(day) = 2007
...
partitions: p_2006,p_2007,p_2008
```

Старайтесь в WHERE использовать те же функции, что и при разбиении на части:

```
EXPLAIN PARTITIONS SELECT * FROM sales_by_day
  WHERE day BETWEEN '2007-01-01' AND '2007-12-31'
...
partitions: p_2007
```

Вопросы?



О чем речь

Университет Текарт

Общие рекомендации I

- Выбираем только нужные колонки и строки, используем внешние ключи для ускорения `JOIN`;
- Chopping up a query: ограничиваем такие выражения, как обновление всей таблицы, с помощью `LIMIT` или другими методами;
- Разбиваем `JOIN` на несколько простых выражений, когда это возможно;
- `mysql_fetch_array` – иллюзия извлечения данных построчно, на самом деле все данные сразу помещаются в память. Используем `mysql_unbuffered_query`, если это возможно.

Общие рекомендации II

- JOIN приводят к вложенным циклам (из-за чего не поддерживаются все типы JOIN) $\Rightarrow$ минимизируем их количество и обходимся без multiple join;
- При выполнении JOIN и подзапросов создаются таблицы в памяти, если при этом есть поля с BLOB или TEXT, то таблица пишется на диск – плохо;
- Замена подзапросов на JOIN часто улучшает производительность, но каждый конкретный случай требует проверки;

Не забываем про query optimizer hints I

- `HIGH_PRIORITY/LOW_PRIORITY` – Устанавливает приоритет для выражения
- `DELAYED` – Используется для `INSERT` и `REPLACE`. Выражение выполняется не сразу, строки помещаются в буфер
- `STRAIGHT_JOIN` – Используется при нескольких `JOIN` для сортировке их в указанном порядке
- `SQL_SMALL_RESULT/SQL_BIG_RESULT` – Используется для `GROUP BY` и `DISTINGS`. Сообщает MySQL стоит ли помещать временную таблицу на диск
- `SQL_BUFFER_RESULT` – Помещает результат в буфер
- `SQL_CACHE/SQL_NO_CACHE` – Сообщает серверу можно ли кешировать выражение

Не забываем про query optimizer hints II

- `SQL_CALC_FOUND_ROWS` – Используется в `LIMIT` запросах. Не смотря на `LIMIT` считавет все строки, количество которых доступно `FOUND_ROWS()`.
- `FOR UPDATE and LOCK IN SHARE MODE` – Используется в `SELECT` запросах, когда вы знает, что после выборки, будите обновлять эти строки. Поддерживается только в InnoDB
- `USE INDEX, IGNORE INDEX, and FORCE INDEX` – “Советует или заставляет” MySQL использовать индексы

Оптимизация COUNT

- считает либо число значений `COUNT(id)`, либо количество строк `COUNT(*)`, второе – быстрее;
- MyISAM очень быстро обрабатывает `COUNT(*)`, но без `WHERE`.
- подсчет количества значений – покрывающие индексы;
- один из тех случаев, когда может помочь вложенное выражение:

```
SELECT COUNT(*) FROM world.city WHERE ID > 5;

SELECT (SELECT COUNT(*) FROM world.city) - COUNT(*)
FROM world.city WHERE ID <= 5;
```

Оптимизация JOIN и подзапросов

- используйте индексы для связи таблиц
- старайтесь указывать в `GROUP BY` или `ORDER BY` только колонки из одной таблицы, чтобы можно было применить индексы;
- синтаксис и выполнение `JOIN` сильно зависит от версии MySQL;
- старайтесь по возможности заменять вложенные выражения на `JOIN`.

Оптимизация GROUP BY и DISTINCT

- используйте индексы – самый результативный способ
- сервер может использовать таблицу в памяти или дисковое пространство в зависимости от размера результата $\Rightarrow$ `SQL_BIG_RESULT` и `SQL_SMALL_RESULT`;
- старайтесь группировать по целочисленным ключам;
- используйте покрывающие индексы:

```
SELECT actor.first_name, actor.last_name, c.cnt
FROM sakila.actor
  INNER JOIN (
     SELECT actor_id, COUNT(*) AS cnt
     FROM sakila.film_actor
     GROUP BY actor_id
  ) AS c USING(actor_id) ;
```

Оптимизация LIMIT и UNION

- больше смещение – меньше производительность;
- большая таблица – покрывающие индексы:

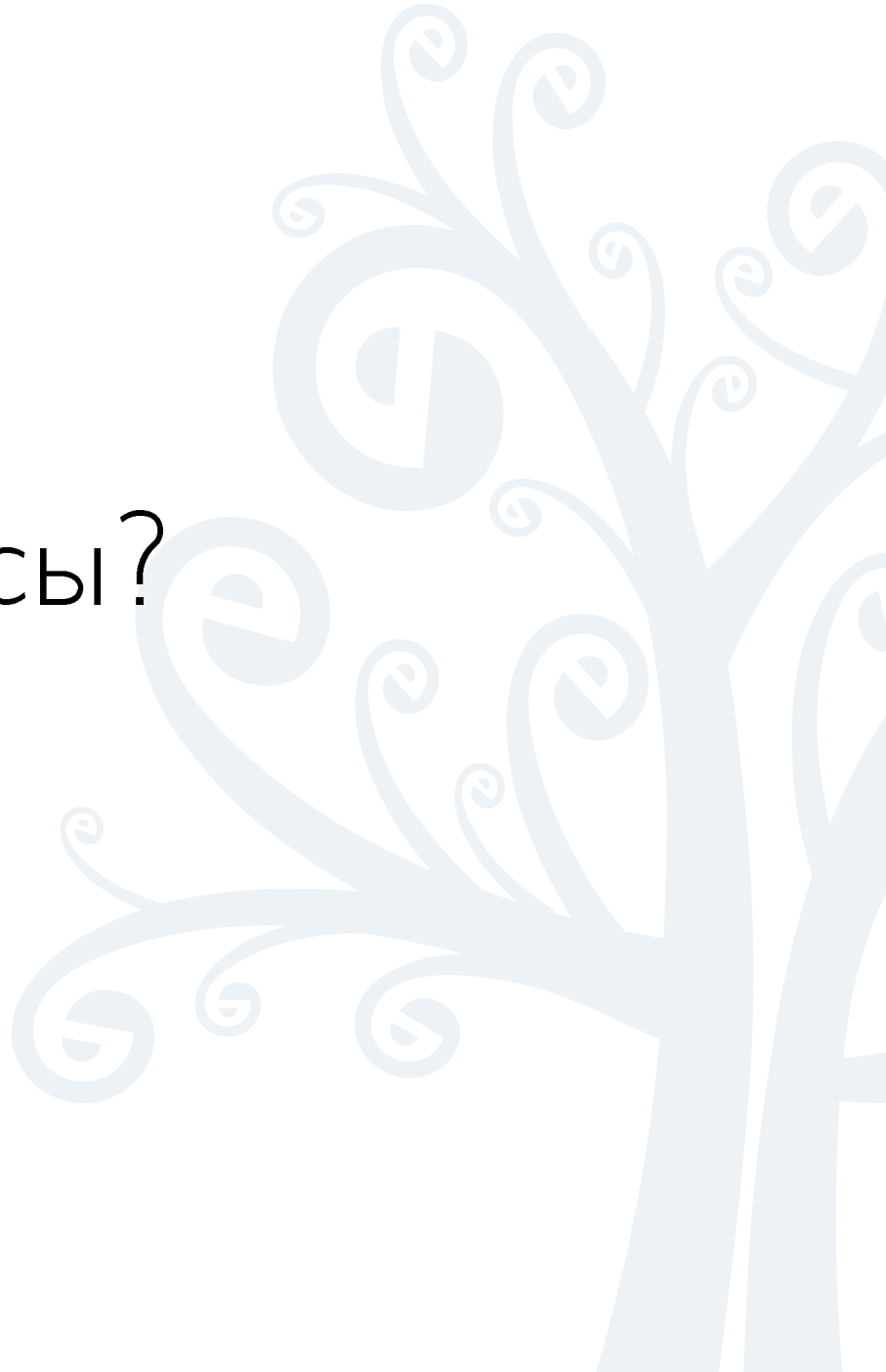
```
SELECT film.film_id, film.description
 FROM sakila.film
   INNER JOIN (
     SELECT film_id FROM sakila.film
     ORDER BY title LIMIT 50, 5
 ) AS lim USING(film_id);
```

- иногда удобно создать специальную колонку с индексом:

```
SELECT film_id, description FROM sakila.film
WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

- аналогично и для UNION, но он всегда создает временную таблицу;
- используйте UNION ALL.

Вопросы?



О чем речь

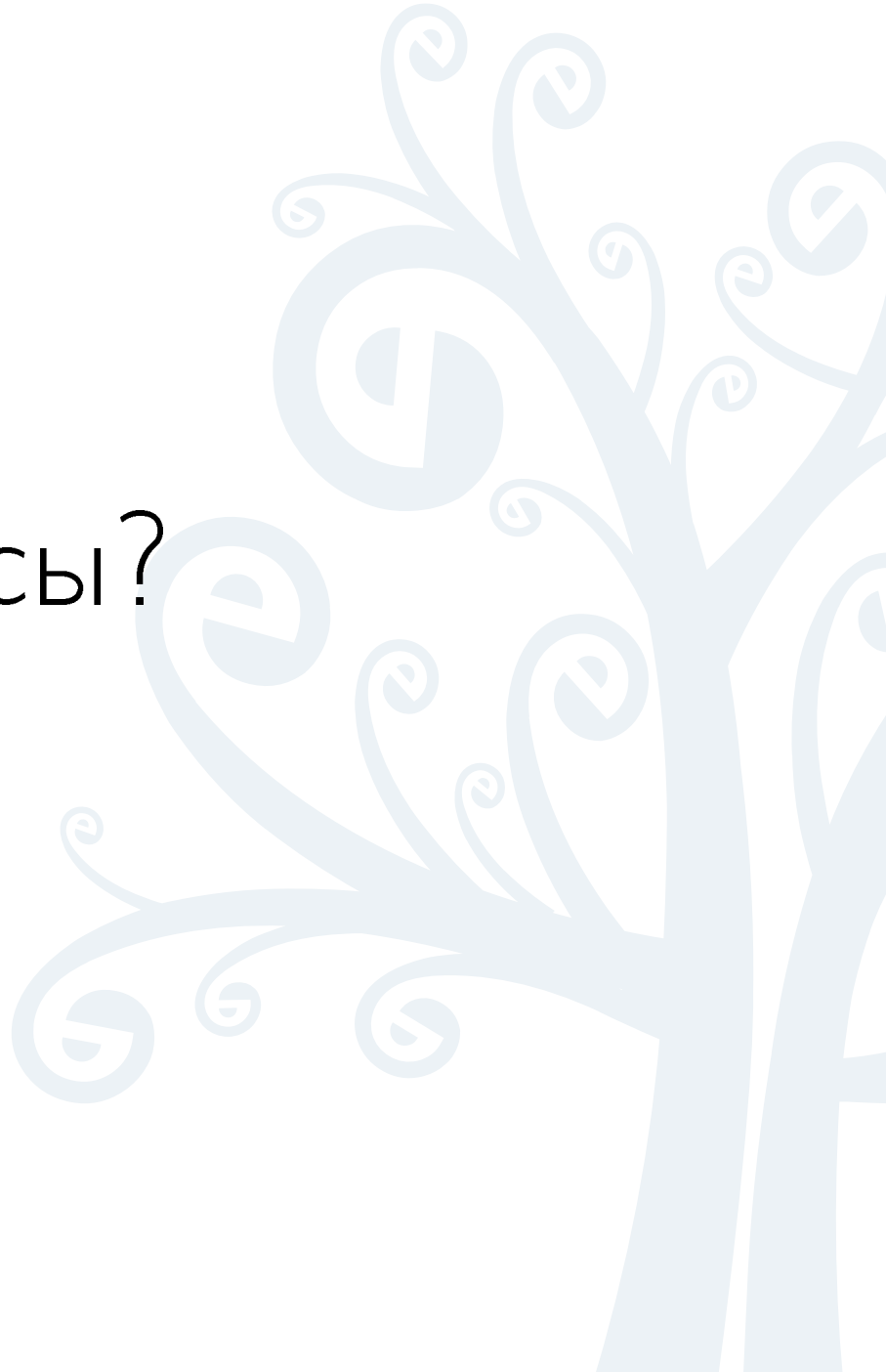
Особенности кеширования

- кешируется весь результат `SELECT`;
- кеш проверяется по полному совпадению SQL выражения, изменения даже в комментариях – перезапись;
- не кешируются: функции типа `NOW()` `CURRENT_DATE()`, пользовательские функции, хранимые процедуры и функции, пользовательские переменные, временные таблицы или любые таблицы с привилегиями на колонки и так далее;
- в InnoDB кеширование осложнено транзакциями;
- тонкое место в процессе кеширования – запись результата в кеш, возможна нехватка памяти и другие сбои.

Рекомендации по кешированию

- самые подходящие выражения для серверного кеширования – которые возвращает мало данных и трудно выполнимы, например `COUNT()`;
- кеширование большого объема данных часто приводит к снижению производительности, в таких случаях – `SQL_NO_CACHE`;
- при активном использовании серверного кеша обратите внимание на такие настройки как `query_cache_type`, `query_cache_size`, `query_cache_min_res_unit`, `query_cache_limit`, `query_cache_wlock_invalidate`;
- используйте клиентский кеш – файловый или MemCache.

Вопросы?



О чем речь

Особенности хранимых процедур и функций

- плохо оптимизируются и плохо кешируются
- хранимые процедуры для несложных и небольших запросов, более сложная логика – вне БД.
- весёлый баг-фича: `ROW_COUNT()` всегда возвращает 1, кроме первой строки `BEFORE` триггера $\Rightarrow$ имитация триггера per-statement вместо `FOR EACH ROW`:

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable FOR EACH ROW
BEGIN
   DECLARE v_row_count INT DEFAULT ROW_COUNT( );
   IF v_row_count <> 1 THEN
      -- Your code here
   END IF;
END;
```

Курсоры & Prepared Statements

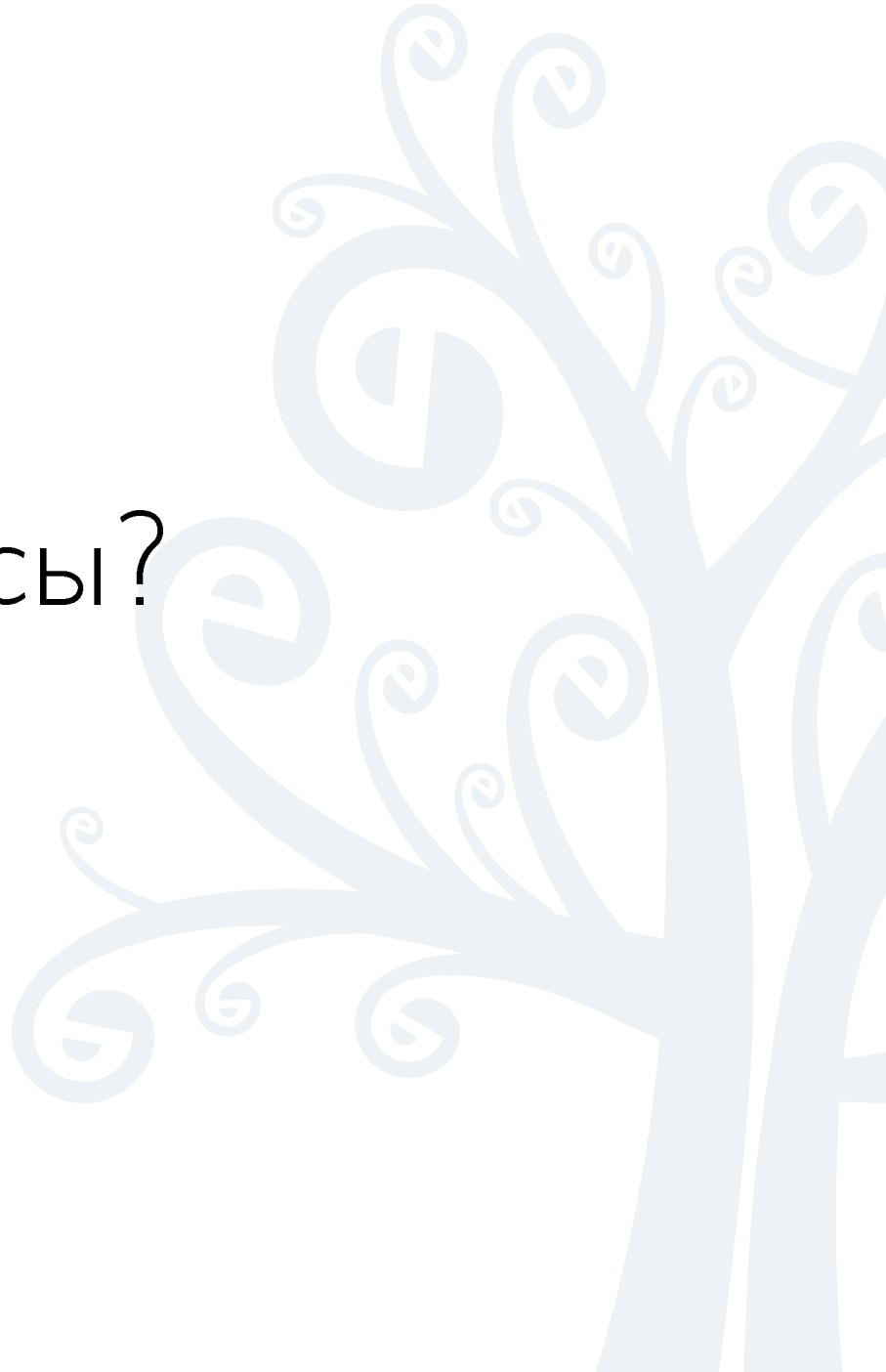
Курсор:

- существует только на чтение и только в прямом порядке;
- выполняет выражение в момент открытия и всегда создает временную таблицу , а это уже известные проблемы;

Prepared Statement:

- парсится и оптимизируется только один раз;
- для передачи параметров создан специальный проток передачи параметров в бинарном виде;
- параметры хранятся в памяти;
- дают прирост производительности при использовании в хранимых процедурах;
- замедляют работу если выполняется один-два раза.

Вопросы?



О чем речь

Полезные сведения

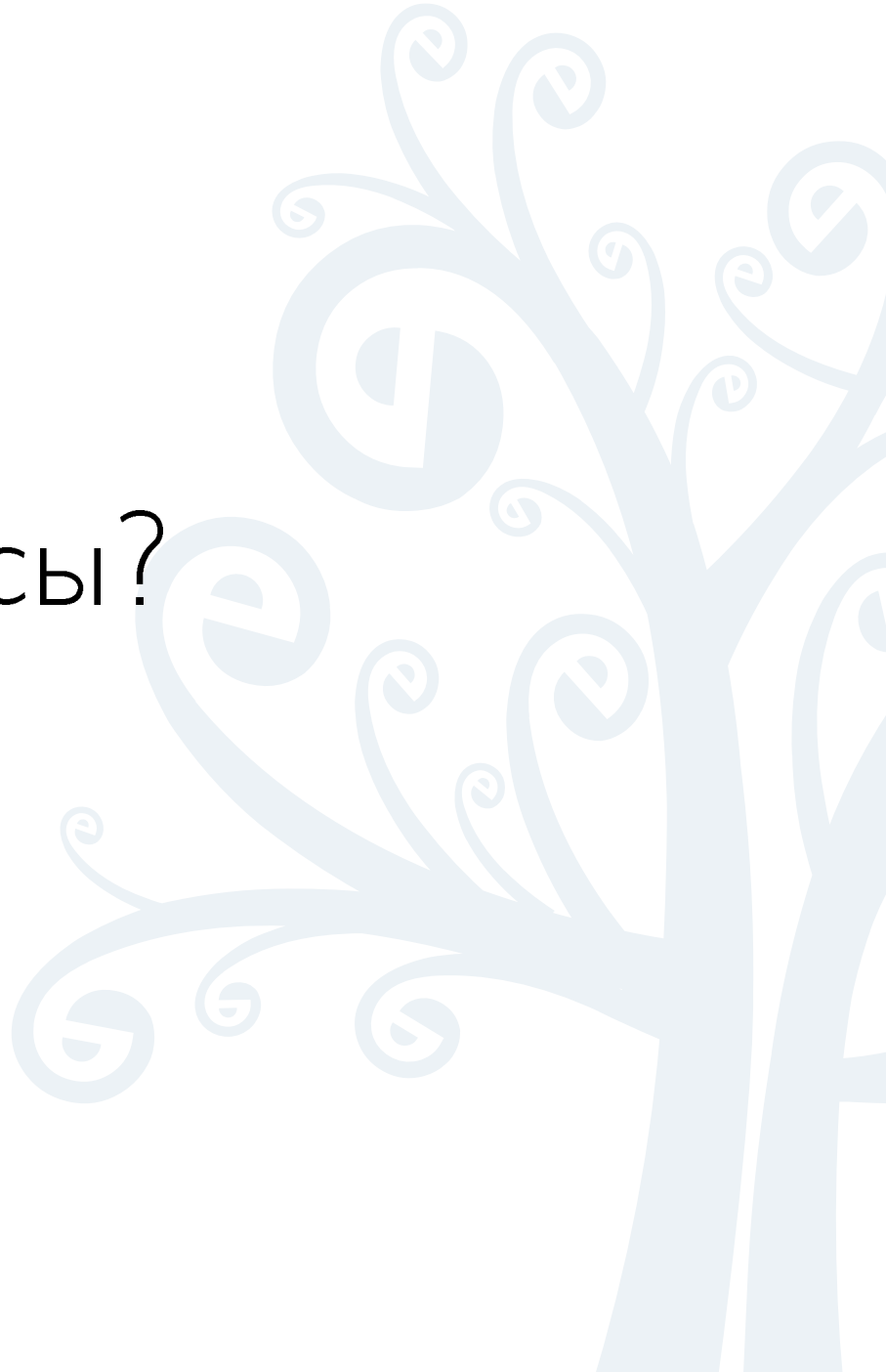
Views может работать в двух режимах – templorary table & merge query
Наиболее интересен второй:

- MySQL объединяет запросы в один
- Намного эффективнее, чем templorary table
- Доступно для обновления

НО

- Такой вариант работает только если выражение для построения View простое.
- Даже очень простое, т.е. не содержит `GROUP BY`, `DISTINCT`, `aggregate functions`, `UNION`, `subqueries`, ...и все что не one to one

Вопросы?



Что читать

- High Performance MySQL: Optimization, Backups, Replication, and More
- MySQL Performance Blog
- MySQL Documentation