

05, май 2016

УДК 004.658.2

Оптимизация схем баз данных и запросов в СУБД *MySQL*

*Периенкова В.Г., студент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»*

*Лукьянченко А.В., студент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»*

*Научный руководитель: Черненький М.В., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»
sunday@nxt.ru*

Анализ производительности

В настоящее время быстро и динамично развиваются веб-приложения, различные системы управления ресурсами предприятия (*ERP*). Большинство из них используют для хранения данных системы управления базами данных. Наиболее известными системами управления реляционными базами данных являются *MySQL*, *PostgreSQL*, *MS SQL Server*, *Oracle 11g*.

В связи с развитием и популяризацией различных социальных сетей, поисковых систем, онлайн-сервисов в глобальной сети Интернет подобные системы вынуждены хранить и обрабатывать миллиарды записей за короткий промежуток времени. Нагрузка популярных автоматизированных информационных систем доходит до сотен тысяч запросов к системе в секунду. Выдерживать такую нагрузку без оптимизации схем баз данных и масштабирования невозможно.

Оптимизация работы с базами данных возможна на двух этапах жизненного цикла системы: на этапе разработки системы и на этапе эксплуатации. На этапе разработки необходимо предусмотреть какой функционал системы будет особенно часто использоваться и в соответствии с этим оптимизировать этот функционал. К примеру, рассмотрим популярный сервис размещения коротких сообщений "*Twitter*": большинство

пользователей только читают сообщения, публикуемые знаменитыми персоналиями. Исходя из этого целесообразно оптимизировать функционал чтения из базы данных.

На этапе эксплуатации возможно вести мониторинг функционала системы и определять части системы, которые имеют проблемы с производительностью. Оптимизация производится непосредственно над уже работающей системой.

Далее будут рассматриваться различные механизмы оптимизации и масштабирования систем, которые использует система управления базами данных (СУБД) *MySQL*: создание индексов, оптимизация запросов к базе данных (БД), оптимизация параметров *MySQL*, шардирование и репликация.

Мониторинг времени выполнения медленных запросов

В СУБД *MySQL* существует механизм для определения долго выполняющихся запросов. Есть возможность записи информации о запросах в файл, которые выполняются дольше заданного промежутка времени.

Для использования такого функционала необходимо использовать глобальные системные переменные *MySQL* *slow_query_log* и *slow_query_log_file*.

С помощью *slow_query_log* можно включить запись медленных запросов, выполнив соответствующий запрос к СУБД: *SET slow_query_log = 1;*

Используя *slow_query_log_file* есть возможность изменить путь до файла, в который будут записаны данные о выполнении медленных запросов. Особенно важным параметром является *long_query_time*, который позволяет задать время в секундах, при превышении которого запрос будет отправлен в лог медленных запросов.

Определение медленно выполняющихся запросов дает ценную информацию, указывая какие части системы подлежат оптимизации. Далее будут рассмотрены различные способы оптимизации медленных запросов.

Определение порядка выполнения запроса

СУБД *MySQL* перед непосредственным выполнением запроса посылает его в оптимизатор (рис. 1). Оптимизатор *MySQL* определяет возможность оптимизации выполнения запроса к базе данных. При наличии такой возможности оптимизатор может изменить порядок выполнения запроса. Однако разработчикам, использующим базу данных, могут быть не очевидны действия оптимизатора и, как следствие, возможен неверный подход к оптимизации. Трудность вызывают обычно запросы, которые затрагивают несколько таблиц. Порядок выборки из таблиц может быть не таким как в запросе и разработчик может поставить неверные индексы.

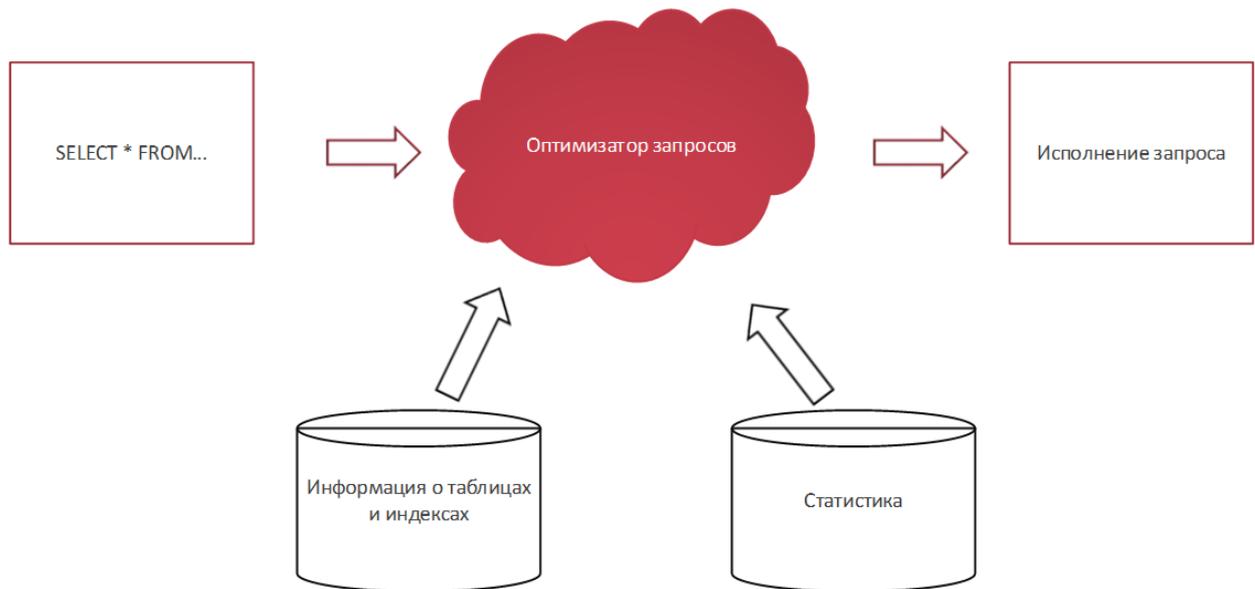


Рис. 1. Оптимизатор запросов *MySQL*

Для определения порядка выполнения запроса существует ключевое слово *EXPLAIN*. Оно позволяет посмотреть порядок выполнения любого запроса выборки данных. Для его использования необходимо написать *EXPLAIN* перед самим запросом. Тогда запрос выполнен не будет, но будет выведена подробная информация о порядке выполнения запроса, используемых и возможных индексах. Рассмотрим пример использования ключевого слова *EXPLAIN* на запрос, затрагивающий данные из двух таблиц:

```
EXPLAIN SELECT * FROM table1 t1 JOIN table2 t2 WHERE t2.amount > 5000;
```

Запрос выведет информацию в виде таблицы, в строках которой по порядку будут указаны в какой последовательности был выполнен запрос, а также дополнительная информация по запросу к каждой таблице:

1. *table* - таблица, к которой относится выводимая строка.
2. *type* - тип связывания, показывает будет ли выполнен полный просмотр таблицы или только части.
3. *possible_keys* - служит для указания индексов, которые может использовать *MySQL* для нахождения строк в этой таблице.
4. *key* - содержит индекс, который будет использован для выборки данных из таблицы.

5. *key_len* - длина используемого ключа. По ней можно определить какая часть составного ключа будет использована.
6. *ref* - показывает какие колонки используются для выборки по индексу
7. *rows* - количество строк, которые MySQL проанализирует в указанной таблице для выполнения запроса
8. *extra* - содержит дополнительную информацию о том, как будет выполняться запрос. Полезна для определения типа сортировки в запросе.

Использование индексов

Создание индексов - один из главных механизмов оптимизации запросов выборки данных из таблиц. Индексы позволяют создать упорядоченные структуры данных для быстрого поиска записей в таблицах. Индекс возможно построить на одну или несколько колонок в пределах одной таблицы. На одну таблицу может быть построено несколько индексов под разные запросы. Однако в запросе может использоваться для одной таблицы только один индекс. Создавая индекс на колонки, *MySQL* создает еще одну индексную таблицу, которая содержит значения колонок, по которым построен индекс в отсортированном виде и указатель на соответствующие строки в таблице, на которую построен индекс.

В MySQL различают индексы следующих типов:

1. *B-tree*
2. *Spatial (R-tree)*
3. *Hash*

Самым распространенным является *B-tree*. Этот индекс оптимален для колонок с хорошим распределением значений и высокой мощностью). При построении индексов *B-tree* типа вводят понятие селективности - число, лежащее в интервале от нуля до единицы, показывающее количество повторений записей в колонке, по которой строится индекс. Селективность вычисляется путем отношения количества уникальных записей на полное количество записей по колонкам, которые входят в индекс.

Spatial ключи используются для индексирования многомерных данных, таких как геолокационные отметки на карте.

Hash индексы могут быть использованы только для таблиц типа *Memory*. Их принцип состоит в использовании специальной хэш функции и хранения возвращаемого ей значения для отображения индекса на данные. Это позволяет сократить размер индексов.

Очевидно, что индексы накладывают дополнительные расходы на диск, частично дублируя информацию из таблиц. Также для построения индексов и их поддержания замедляются операции добавления и удаления данных в таблицах. Исходя из этого индексы целесообразно строить на таблицы, операции к которым в основном состоят из чтения.

Для построения *B-tree* индекса в MySQL необходимо выполнить соответствующий запрос:

```
CREATE INDEX index_name ON table (name);
```

Этот запрос создает индекс с именем *index_name* на таблицу *table* по колонке *name*. Для создания составного индекса по нескольким колонкам необходимо указать колонки в круглых скобках через запятую.

Также существует понятие кластерного индекса, который составлен по правилам *B-tree* индекса, однако хранит не указатель на строки в таблице, а непосредственно все данные таблицы. MySQL решает самостоятельно какие индексы следует хранить подобным образом. Примером кластерного индекса в MySQL является первичный ключ.

Оптимизация запросов

Безусловно, оптимизация схемы является одним из наиболее важных компонентов, помогающих достичь наибольшую производительность. Но при этом нельзя забывать, что плохо сконструированный запрос к хорошо оптимизированной схеме может выполняться очень медленно.

Основной причиной медленного выполнения запросов является большой объем хранимых данных. Безусловно, на объем хранимых данных повлиять мы не можем, но сконструировать запрос таким образом, чтобы он обращался к меньшему количеству данных, возможно.

При построении любого запроса к MySQL необходимо руководствоваться следующими правилами:

1. Избавьтесь от выбора ненужных строк. Например, Вы хотите выбрать 10 лучших студентов на потоке. При этом выбираете всех студентов, сортируя их по успеваемости, и отрезаете нужное Вам количество. Вместо этого лучше включить запрос выражение *LIMIT* для ограничения количества кортежей, возвращенных командой *SELECT*.

2. Выбирайте только нужные столбцы из соединения нескольких таблиц. Если вы используете в своих запросах операции соединения, то необходимо точно представлять

какие данные и из каких Вам необходимо получить. Очень редко необходимо извлечь данные из всех столбцов нескольких таблиц.

3. Остерегайтесь ‘SELECT *’. Выборка всех столбцов редко является необходимой и препятствует использованию покрывающих индексов, что в свою очередь увеличивает использование ресурсов сервера.

Однако, если Ваш запрос выбирает исключительно необходимые данные, но по-прежнему выполняется достаточно долго, то необходимо проанализировать тип доступа к запрашиваемым данным. Используя оператор *EXPLAIN* вы можете увидеть какой тип доступа используется в данном запросе.

Например:

```
mysql> EXPLAIN SELECT * FROM test_db.student WHERE record_book = 1234\G
```

```
***** 1. row *****
```

```
id: 1
```

```
select_type: SIMPLE
```

```
table: student
```

```
type: ALL
```

```
possible_keys: NULL
```

```
key: NULL
```

```
key_len: NULL
```

```
ref: NULL
```

```
rows: 5037
```

```
Extra: USING WHERE
```

Здесь *type = ALL* свидетельствует о том, что ни один из индексов не будет использован в данном запросе и, соответственно, будет просмотрено 5000 записей. При наличии подходящего индекса *idx_record_book* количество просмотренных записей заметно уменьшается:

```
***** 1. row *****
```

```
id: 1
```

```
select_type: SIMPLE
```

```
table: student
```

```
type: ref
```

```
possible_keys: idx_record_book
```

```
key: idx_record_book
```

```
key_len: 4
```

ref: const

rows: 256

Extra: NULL

Этот пример наглядно демонстрирует важную роль индексов. Они позволяют применять при обработке запроса наиболее эффективный тип доступа и анализировать лишь те строки, которые необходимы.

Разделяй и властвуй. Еще один эффективный способ оптимизировать запрос - разбить его на несколько более простых запросов. Хорошим примером такой оптимизации является необходимая операция удаления данных из СУБД. Например, нам необходимо удалить данные обо всех студентах, которые закончили обучение более 5 лет назад из университета. Есть два способа это сделать:

1. Постараться одним запросом удалить все данные

```
mysql> DELETE FROM test_db.student WHERE student.graduation_date <
DATE_SUB(NOW(), INTERVAL 5 YEAR);
```

Если данному запросу удовлетворяет большое количество записей, то он будет не только долго выполняться, но и замедлит работу всей системы в целом.

2. Разбить требуемую операцию на несколько подзапросов и выполнить их несколько раз. Также рекомендуется создать искусственную задержку между данными операциями для снижения нагрузки на сервер.

```
mysql> DELETE FROM test_db.student WHERE student.graduation_date <
DATE_SUB(NOW(), INTERVAL 5 YEAR) LIMIT 5000;
```

Помимо вышеперечисленных способов оптимизации запросов, *MySQL* предоставляет большой инструментарий для оптимизации запросов конкретных типов, а также дает возможность подсказывать оптимизатору план выполнения запроса. Однако стоит заметить, что данные возможности СУБД стоит использовать с большой осторожностью, так как они требуют полного понимания работы оптимизатора *MySQL*.

Оптимизация параметров *MySQL*

Оптимальная настройка сервера сильно разнится в зависимости от реализуемых задач. Лучший вариант конфигурации может быть определен эмпирическим путем.

Перед настройкой сервера стоит подготовить комплект тестов, отражающих как эталонную нагрузку, так и ее пограничные случаи (сложные и медленные запросы). После обнаружения и устранения определенной проблемы (проблемного запроса) необходимо повторно протестировать систему, так как подобное вмешательство может негативно повлиять на остальную систему в целом.

Именно такой подход к оптимизации параметров, когда после каждой нововведённой оптимизации вы тестируете систему заново, является наиболее верным, так как очевидно, что снижение времени выполнения одного запроса может привести к увеличению времени выполнения других. При этом стоит помнить о важности составленных Вами тестов и о том, что база может дополняться и расширяться, тем самым требуя корректировки конфигурационных параметров.

Масштабирование

Оптимизация схем баз данных, создание индексов, корректировка запросов, денормализация для увеличения скорости выборки данных дают несомненно большой прирост производительности. Однако в больших проектах этого может не хватить. Тогда необходимо горизонтально масштабировать базы на несколько физических машин. Это возможно с помощью двух известных механизмов: шардирование и репликация.

Масштабирование - одна из самых сложных задач при росте системы. На этапе разработки невозможно точно предугадать какие мощности будут необходимы для поддержания работоспособности системы. Поэтому обычно горизонтальное масштабирование проводят когда система находится в работе.

Для определения значения нагрузки, которую может выдержать система проводят нагрузочное тестирование. Если горизонтальное масштабирование всё же необходимо, то существуют несколько механизмов для его реализации.

Первым рассмотрим шардирование. Используя шардирование, база разделяется на несколько частей по определенному критерию и каждая из частей (шард) помещается на отдельный сервер. Это позволяет снизить нагрузку с каждого сервера примерно в n раз, где n равен числу серверов, на которых происходит шардирование. Пример шардирования приведен на рис. 2.

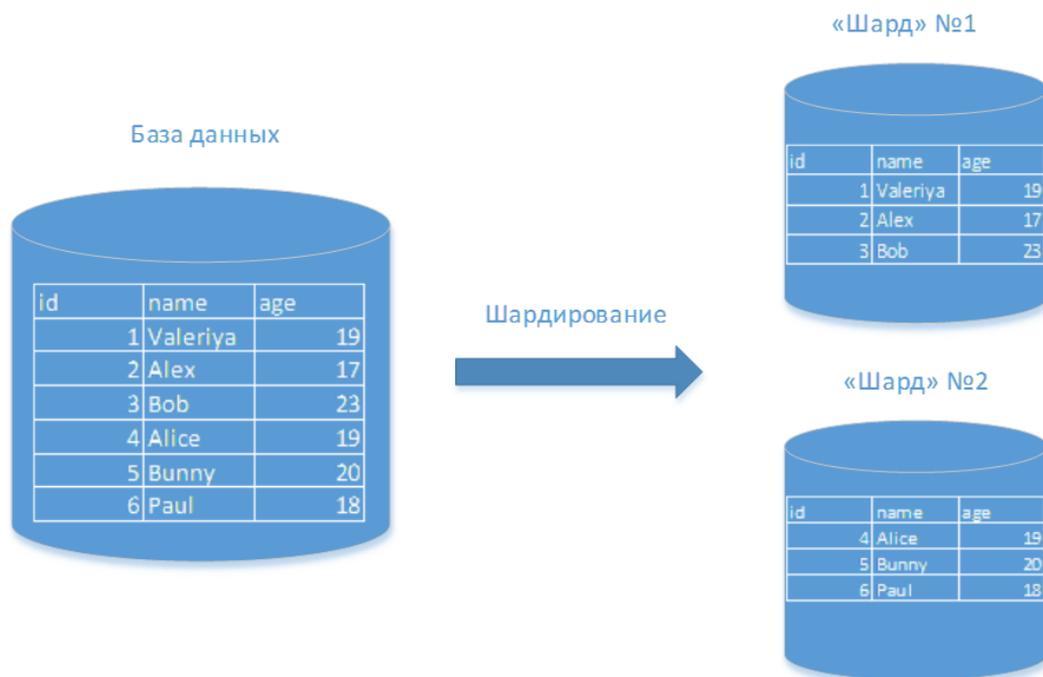


Рис. 2. Пример шардирования

Репликация определяет несколько иной подход. В стандартном случае создаются несколько серверов, среди которых есть ведущий и один или несколько ведомых (рис. 3).

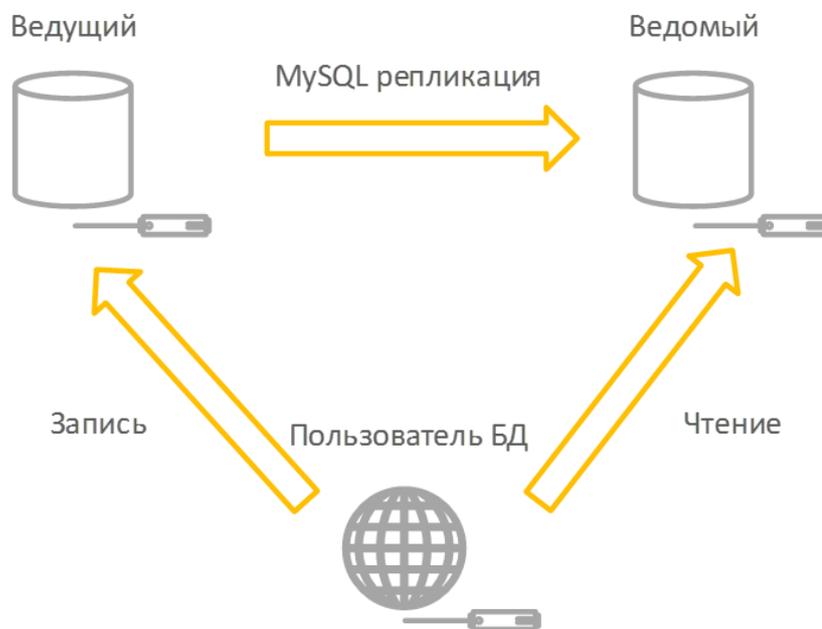


Рис. 3. Репликация БД MySQL

Все операции чтения и удаления производятся на ведущем сервере, а операции чтения с ведомых серверов. Ведомые серверы являются копиями ведущего сервера и постоянно считывают с него новые изменения. Это позволяет распределить нагрузку на операции чтения на несколько серверов.

Список литературы

- [1] Schwartz B., Zaitsev P., Trachenko V., Zawodny J., Lentz A., Balling D. High performance MySQL. 3rd ed. O'Reilly Media Inc., 2012. 826 p.
- [2] DuBois P. MySQL Cookbook: Solutions for Database Developers and Administrators. 3rd Edition. O'Reilly Media Inc., 2014. 866 p.
- [3] Молинаро Э. SQL. Сборник рецептов: пер. с англ. СПб: Символ-Плюс, 2009. 672 с. [Molinaro A. SQL Cookbook. O'Reilly Media Inc., 2006.].
- [4] MySQL 5.7 Reference Manual. Режим доступа: <http://dev.mysql.com/doc/refman/5.7/en/> (дата обращения 21.02.2016).
- [5] Малакшинов С. Обзор типов индексов Oracle, MySQL, PostgreSQL, MS SQL. Режим доступа: <https://habrahabr.ru/post/102785/> (дата обращения 21.02.2016).
- [6] Индексы в MySQL. Режим доступа: <http://ruhighload.com/post/Работа+с+индексами+в+MySQL> (дата обращения 21.02.2016).
- [7] Шардинг и репликация. Режим доступа: <http://ruhighload.com/index.php/2009/05/06/шардинг-партиционирование-репликац/> (дата обращения 21.02.2016).