Урок 4

Углубленное проектирование реляционных БД

Продолжение знакомства с симбиозом MySQL и PHP. Понятие нормализации данных, различные формы нормализации. Связи в БД. Резервное копирование данных и оптимизация производительности БД. Знакомство с механизмом транзакций.

На курсе «PHP. Level 1» мы познакомились с общими принципами работы с БД.

Тщательное проектирование базы данных важно для корректной работы приложения. Как установка принтера в дальнем конце офиса ведет к снижению производительности труда, так и размещение данных со слабыми взаимосвязями снижает эффективность ПО, и может вынудить сервер БД тратить значительное время на поиск требуемых данных. Разрабатывая структуру БД, задумайтесь о вопросах, возникающих при работе с ней. Например, какие дополнительные сведения имеются о продаваемом продукте, верны ли имя пользователя и пароль.

Реляционные БД

MySQL – это реляционная база данных. Важная особенность реляционных систем, их отличие от одноуровневых баз данных – возможность располагать данные в нескольких таблицах. Взаимосвязанные данные можно хранить в отдельных таблицах и объединять по ключу, общему для обеих таблиц.

Ключ – это отношение (**relation**) между таблицами. Выбор первичного ключа (**primary key**) – наиболее важное решение, принимаемое при разработке новой базы данных. Главное – гарантировать уникальность выбранного ключа. Если есть вероятность того, что значение атрибута может совпадать у двух записей, его нельзя использовать в качестве первичного ключа. Если таблица содержит ключевые поля из другой таблицы, между ними образуется связь (взаимоотношением внешнего ключа – **foreign key**) – например, «начальник-подчиненный» или «покупатель-покупка».

В качестве примера создадим таблицу в базе данных **Customers**, соответствующую книжному интернет-магазину. Выполним следующий SQL-код для создания таблицы и вставки данных:

```
CREATE TABLE Customers (
    id INT NOT NULL AUTO_INCREMENT,
    firstname VARCHAR(32),
    secondname VARCHAR(50),
    adress VARCHAR(256),
    telephone VARCHAR(20),
    bookTitle VARCHAR(256),
    bookAuthor1 VARCHAR(64),
    bookAuthor2 VARCHAR(64),
    pageCount INT(4),
    dateOrder DATETIME,
PRIMARY KEY(id)) CHARACTER SET utf8;
```

```
INSERT INTO Customers VALUES
      (1, 'Александр', 'Иванов', 'Ленинский проспект 68 - 34, Москва 119296',
'+7-920-123-45-67', 'Золотые сказки', 'Александр Сергеевич Пушкин', '', 128,
'2013-04-18 14:56:00'),
      (NULL, 'Дмитрий', 'Петров', 'Хавская 3 - 128, Москва 115162',
'+7-495-123-45-67', 'ASP.NET MVC 4', 'Джесс Чедвик', 'Тодд Снайдер', 432,
'2013-02-11 09:18:00'),
```

```
      (NULL, 'Дмитрий', 'Петров', 'Хавская 3 - 128, Москва 115162',
'+7-495-123-45-67', 'LINQ. Язык интегрированных запросов', 'Адам Фримен',
'Джозеф С. Раттц', 656, '2013-02-25 19:44:00'),
      (NULL, 'Александр', 'Иванов', 'Ленинский проспект 68 - 34, Москва
119296', '+7-920-123-45-67', 'Сказки Старого Вильнюса', 'Макс Фрай', '', 480,
'2013-05-02 14:12:00'),
      (NULL, 'Александр', 'Иванов', 'Ленинский проспект 68 - 34, Москва
119296', '+7-920-123-45-67', 'Реверс', 'Сергей Лукьяненко', 'Александр Громов',
352, '2013-03-12 08:25:00'),
      (NULL, 'Елена', 'Козлова', 'Тамбовская - 47, Санкт-Петербург 192007',
'+7-920-765-43-21', 'Золотые сказки', 'Александр Сергеевич Пушкин', '', 128,
'2013-04-12 12:56:00'),
      (NULL, 'Елена', 'Козлова', 'Тамбовская - 47, Санкт-Петербург 192007',
'+7-920-765-43-21', 'ASP.NET MVC 4', 'Джесс Чедвик', 'Тодд Снайдер', 432,
'2013-04-14 10:11:00');
```

Теперь, когда у нас есть отдельные таблицы для хранения взаимосвязанных данных, нужно подумать об элементах в каждой таблице, которые будут описывать связи.

Нормализация

Представление о взаимоотношениях данных и наиболее эффективном способе их организации называется **нормализацией**. Она заключается в разделении данных на основе логических взаимоотношений, чтобы минимизировать дублирование. Повторяющиеся данные понапрасну расходуют дисковое пространство сервера и затрудняют обслуживание. При внесении изменений в повторяющиеся данные есть риск пропустить какие-то из них, что может привести к возникновению несогласованностей в БД.

Но лучшее – враг хорошего: когда данные хранятся по частям в отдельных таблицах, это может потребовать слишком больших накладных расходов на их извлечение, да и запросы могут получаться чересчур замысловатыми. Главная цель – найти золотую середину.

Вернемся к примеру с книжным интернет-магазином. Сайт должен хранить данные о покупателях, включая имя и фамилию пользователя, адрес и номер телефона, а также информацию о книгах, включая название, автора, количество страниц и дату продажи каждого экземпляра. Изначально мы разместили всю информацию в одной таблице:

id	firstname	secondname	adress	telephone	bookTitle	bookAuthor1	bookAuthor2	pageCount	dateOrder
1	Александр	Иванов	Ленинский проспект 68 - 34, Москва 119296	+7-920-123-45-67	Золотые сказки	Александр Сергеевич Пушкин		128	2013-04-18 14:56:00
2	Дмитрий	Петров	Хавская 3 - 128, Москва 115162	+7-495-123-45-67	ASP.NET MVC 4	Джесс Чедвик	Тодд Снайдер	432	2013-02-11 09:18:00
3	Дмитрий	Петров	Хавская 3 - 128, Москва 115162	+7-495-123-45-67	LINQ. Язык интегрированных запросов	Адам Фримен	Джозеф С. Раттц	656	2013-02-25 19:44:00
4	Александр	Иванов	Ленинский проспект 68 - 34, Москва 119296	+7-920-123-45-67	Сказки Старого Вильнюса	Макс Фрай		480	2013-05-02 14:12:00
5	Александр	Иванов	Ленинский проспект 68 - 34, Москва 119296	+7-920-123-45-67	Реверс	Сергей Лукьяненко	Александр Громов	352	2013-03-12 08:25:00
6	Елена	Козлова	Тамбовская - 47, Санкт-Петербург 192007	+7-920-765-43-21	Золотые сказки	Александр Сергеевич Пушкин		128	2013-04-12 12:56:00
7	Елена	Козлова	Тамбовская - 47, Санкт-Петербург 192007	+7-920-765-43-21	ASP.NET MVC 4	Джесс Чедвик	Тодд Снайдер	432	2013-04-14 10:11:00

Размещение всех данных в одной таблице может показаться заманчивым, но приводит к напрасному расходованию пространства в базе данных и делает утомительной операцию обновления. С каждой новой покупкой все сведения о покупателе записываются повторно. Для каждой книги можно указать не более двух авторов. Кроме того, если покупатель поменяет адрес, это потребует внести изменения в каждую связанную с ним запись.

Формы нормализации

Процесс нормализации состоит из трех этапов – форм. Первый – приведение к первой нормальной форме, должен быть выполнен перед приведением базы данных ко второй нормальной форме. Аналогично, невозможно привести базу данных к третьей нормальной форме, минуя вторую. Процесс нормализации приводит структуру данных в соответствие с тремя нормальными формами.

Первая нормальная форма

Необходимо, чтобы приведенная к первой нормальной форме база данных соответствовала трем требованиям. Ни одна таблица не должна иметь повторяющихся столбцов, содержащих одинаковые по смыслу значения, и все столбцы должны содержать единственное значение. Обязательно должен быть определен первичный ключ, который уникальным образом описывал бы каждую строку. Это может быть один столбец или комбинация из нескольких – в зависимости от того, сколько потребуется столбцов для обеспечения уникальной идентификации строк.

В созданной нами таблице нарушено требование, предъявляемое к повторяющимся столбцам, потому что в столбцах «**bookAuthor1**» и «**bookAuthor2**» хранятся одинаковые по смыслу данные. Это несоответствие надо устранить, иначе может потребоваться добавить много полей для хранения имен авторов (например, если у книги три автора), что приведет к неоправданному расходу пространства. Или может не хватить предусмотренного количества полей для хранения всех имен, если над книгой трудилось много авторов. Решение – переместить имена всех авторов в отдельную таблицу, которая будет связана с таблицей книг.

Вторая нормальная форма

Первая нормальная форма снижает избыточность данных в строке. Вторая же ликвидирует избыточность данных в столбцах. Нормальные формы получаются последовательно. Для приведения ко второй нормальной форме необходимо, чтобы таблицы уже соответствовали требованиям первой.

Чтобы привести таблицу базы данных ко второй нормальной форме, нужно определить, какие из ее столбцов содержат одни и те же данные для нескольких строк. Такие столбцы нужно поместить в отдельную таблицу, связав ее с первоначальной по ключу. Другими словами, нужно отыскать поля, не зависящие от первичного ключа. Имена авторов и такие сведения о книгах, как количество страниц, никак не связаны с первичным ключом, идентифицирующим покупателя. Выделим эту информацию в

отдельные таблицы (это действие будет включать в себя также нормализацию по первой форме, т.к. мы выделяем в отдельную таблицу имена авторов):

```
-- Создадим новую таблицу Books
CREATE TABLE Books (
    bookId INT NOT NULL AUTO_INCREMENT,
    title VARCHAR(500),
    authors VARCHAR(1000),
    pageCount INT(4),
PRIMARY KEY(bookId)) CHARACTER SET utf8;
```

```
-- Заполним таблицу Books и столбец bookId таблицы customers
INSERT INTO Books VALUES(1, 'Золотые сказки', 'Александр Сергеевич Пушкин', 128),
      (NULL, 'ASP.NET MVC 4', 'Джесс Чедвик, Тодд Снайдер', 432),
      (NULL, 'LINQ. Язык интегрированных запросов', 'Адам Фримен, Джозеф С. Раттц', 656),
      (NULL, 'Сказки Старого Вильнюса', 'Макс Фрай', 480),
      (NULL, 'Реверс', 'Сергей Лукьяненко, Александр Громов', 352);
```

```
-- Видоизменим таблицу Customers, удалив избыточные столбцы bookTitle, bookAuthor1, bookAuthor2, pageCount
-- и добавим столбец bookId
ALTER TABLE customers DROP bookTitle, DROP bookAuthor1, DROP bookAuthor2, DROP pageCount, ADD bookId INT(4);
```

```
UPDATE Customers SET bookId = 1 WHERE id = 1;
UPDATE Customers SET bookId = 2 WHERE id = 2;
UPDATE Customers SET bookId = 3 WHERE id = 3;
UPDATE Customers SET bookId = 4 WHERE id = 4;
UPDATE Customers SET bookId = 5 WHERE id = 5;
UPDATE Customers SET bookId = 1 WHERE id = 6;
UPDATE Customers SET bookId = 2 WHERE id = 7;
```

Этот код позволяет создать дополнительную таблицу **Books**, хранящую данные о книгах. Но полная нормализация по второй форме еще не выполнена. Можно заметить, что в нескольких строках таблицы **Customers** повторяется информация о пользователях, сделавших несколько заказов. Для приведения ко второй нормальной форме мы определим новую таблицу **Orders** (Заказы):

```
-- Создадим новую таблицу Orders (orderId - идентификатор заказа, userId - идентификатор пользователя, который сделал заказ)
CREATE TABLE Orders (
    orderId INT NOT NULL AUTO_INCREMENT,
    userId INT,
    bookId INT,
    dateOrder DATETIME,
PRIMARY KEY(orderId)) CHARACTER SET utf8;
```

```
-- Видоизменим таблицу Customers и добавим данные Orders. Обратите внимание,
-- что проводить нормализацию заполненной базы данных - трудоемкая задача,
-- поэтому ее нужно проводить на этапе проектирования базы данных
INSERT INTO Orders (dateOrder, bookId) SELECT dateOrder, bookId FROM Customers;
```

```
UPDATE Orders SET userId = 1 WHERE orderId = 1;
UPDATE Orders SET userId = 2 WHERE orderId = 2;
UPDATE Orders SET userId = 2 WHERE orderId = 3;
UPDATE Orders SET userId = 1 WHERE orderId = 4;
UPDATE Orders SET userId = 1 WHERE orderId = 5;
UPDATE Orders SET userId = 3 WHERE orderId = 6;
UPDATE Orders SET userId = 3 WHERE orderId = 7;
```

```
ALTER TABLE customers DROP dateOrder;
```

```
-- Удалить дублирующие данные пользователей из таблицы Customers
ALTER IGNORE TABLE customers ADD UNIQUE INDEX (telephone);
ALTER TABLE customers DROP bookId;
```

Теперь данные оформлены безупречно. У нас появились отдельные таблицы со сведениями о покупателях (**Customers**), книгах (**Books**) и покупках (**Orders**).

Третья нормальная форма

Если приведение к первой и второй нормальным формам завершено, возможно, не потребуется больше ничего делать с базой данных, чтобы привести ее к третьей нормальной форме. Для этого нужно просмотреть таблицы и выделить данные, которые не зависят от первичного ключа, но зависят от других значений.

В таблице **Customers** компоненты адреса не имеют прямого отношения к покупателю. Название улицы и номер дома связаны с почтовым индексом, почтовый индекс – с городом и, наконец, сам город – с областью или краем. Третья нормальная форма требует, чтобы каждая такая часть данных была выделена в отдельную таблицу.

Ниже показано, как можно разделить информацию об адресе, создав отдельную таблицу **Addresses**:

```
-- Создадим новую таблицу Addresses
CREATE TABLE Addresses (
    userId INT NOT NULL AUTO_INCREMENT,
    city VARCHAR(30),
    street VARCHAR(50),
    postcode INT(6),
PRIMARY KEY(userId)) CHARACTER SET utf8;
```

```
-- Видоизменим таблицу Customers и добавим данные в Addresses
ALTER TABLE Customers DROP address;
```

```
INSERT INTO Adresses (city, street, postcode) VALUES
('Москва', 'Ленинский проспект 68 - 34',  119296),
('Москва', 'Хавская 3 - 128',  115162),
('Санкт-Петербург', 'Тамбовская - 47', 192007);
UPDATE Customers SET id = 3 WHERE id = 6;
```

С практической точки зрения можно заметить, что после приведения к третьей нормальной форме было создано больше таблиц, чем хотелось бы иметь в базе данных. Поэтому вы должны сами решать, когда остановить процесс нормализации. Хорошо, если ваши данные будут соответствовать, по крайней мере, второй нормальной форме. Цель – избежать избыточности данных, предотвратить их повреждение и минимизировать занимаемое данными пространство. Кроме того, нужно убедиться, что одни и те же значения не хранятся в нескольких местах. В противном случае, когда эти данные изменятся, вам придется обновлять их везде, что может привести к повреждению базы данных.

Третья нормальная форма еще более снижает избыточность данных, но ценой простоты их представления и производительности. В нашем примере не приходится ожидать, что информация об адресах будет часто изменяться. Однако третья нормальная форма позволяет снизить риск появления орфографических ошибок в названиях городов и улиц. Поскольку это ваша база данных, вам и определять соотношение между нормализацией, простотой и производительностью.

Типы связей

Взаимоотношения, или связи, в базах данных подразделяются на следующие категории:

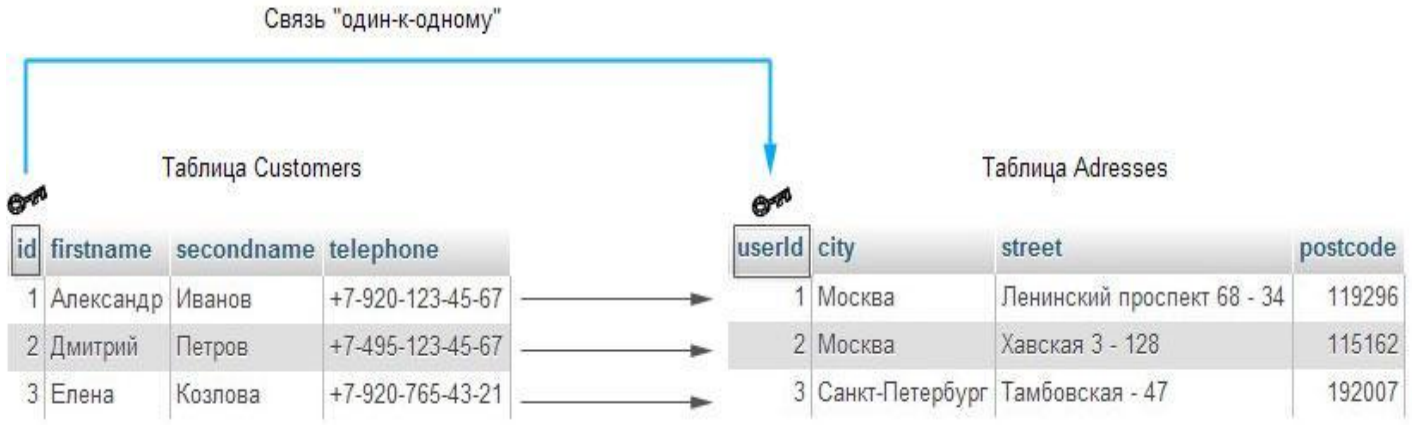
- связи «один к одному»;
- связи «один ко многим»;
- связи «многие ко многим».

Рассмотрим каждую из этих связей на примере созданной нами БД.

Связи «один к одному»

При связи «один к одному» каждому элементу соответствует только один другой элемент. Например, в контексте книжного интернет-магазина эта связь существует между покупателем и адресом доставки.

Каждый покупатель должен иметь единственный адрес доставки. Знак ключа рядом с каждой из таблиц на рисунке ниже указывает на поле, которое является ключом для этой таблицы:



id	firstname	secondname	telephone
1	Александр	Иванов	+7-920-123-45-67
2	Дмитрий	Петров	+7-495-123-45-67
3	Елена	Козлова	+7-920-765-43-21

userId	city	street	postcode
1	Москва	Ленинский проспект 68 - 34	119296
2	Москва	Хавская 3 - 128	115162
3	Санкт-Петербург	Тамбовская - 47	192007

Например, чтобы вывести адрес пользователя «Александр Иванов», можно воспользоваться следующей SQL-конструкцией:

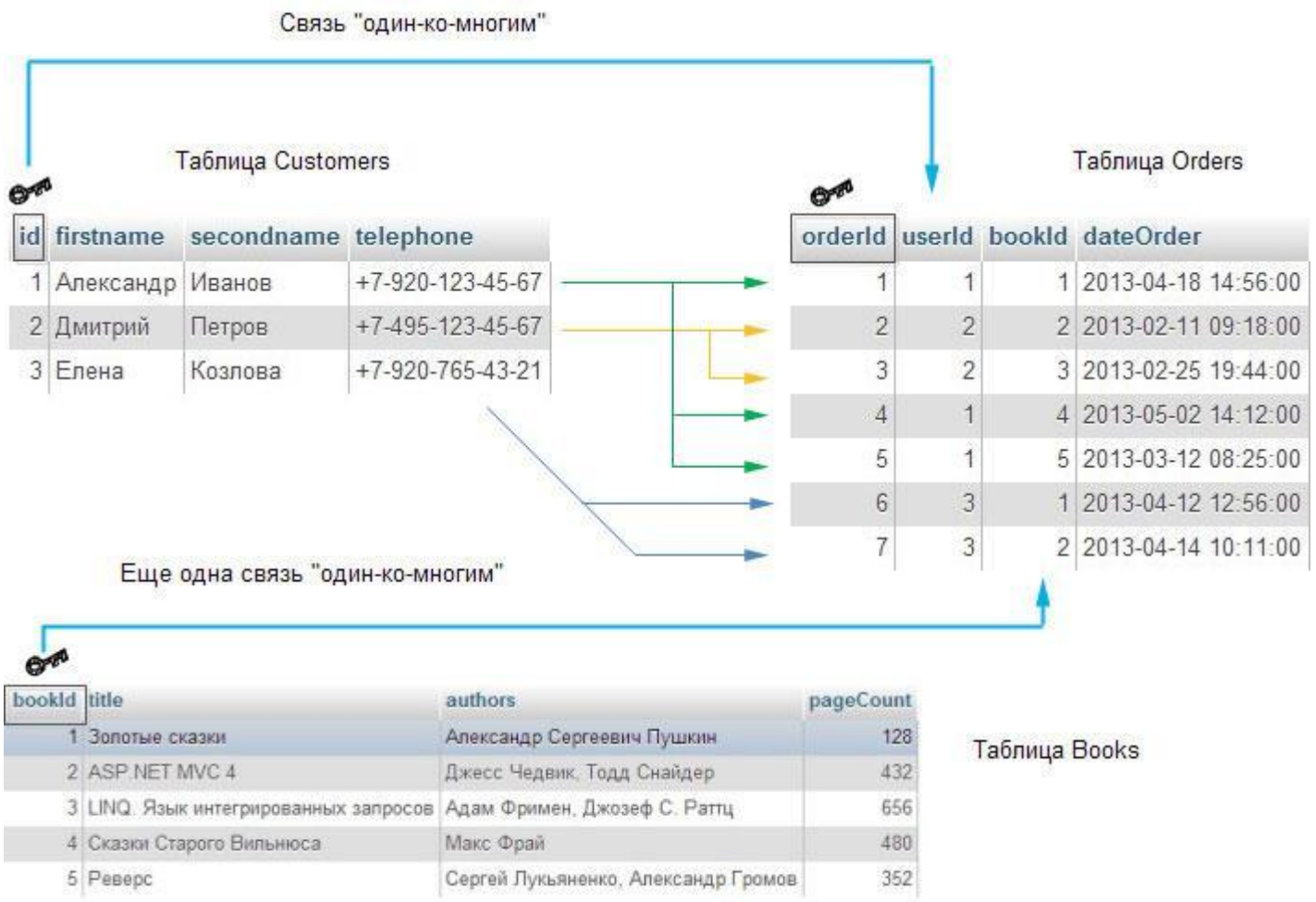
```
SELECT * FROM customers JOIN adresses ON (customers.id = adresses.userid) WHERE customers.id = 1;
```

+ Параметры

id	firstname	secondname	telephone	userId	city	street	postcode
1	Александр	Иванов	+7-920-123-45-67	1	Москва	Ленинский проспект 68 - 34	119296

Связь «один ко многим»

В случае связи «один ко многим» каждый ключ из одной таблицы может встречаться несколько раз в другой. Это наиболее распространенный тип связи. Например, у одного покупателя может быть несколько заказов, и каждый заказ имеет свой уникальный идентификатор, но два покупателя могут заказать одну и ту же книгу:



id	firstname	secondname	telephone
1	Александр	Иванов	+7-920-123-45-67
2	Дмитрий	Петров	+7-495-123-45-67
3	Елена	Козлова	+7-920-765-43-21

orderId	userId	bookId	dateOrder
1	1	1	2013-04-18 14:56:00
2	2	2	2013-02-11 09:18:00
3	2	3	2013-02-25 19:44:00
4	1	4	2013-05-02 14:12:00
5	1	5	2013-03-12 08:25:00
6	3	1	2013-04-12 12:56:00
7	3	2	2013-04-14 10:11:00

bookId	title	authors	pageCount
1	Золотые сказки	Александр Сергеевич Пушкин	128
2	ASP.NET MVC 4	Джесс Чедвик, Тодд Снайдер	432
3	LINQ. Язык интегрированных запросов	Адам Фримен, Джозеф С. Раттц	656
4	Сказки Старого Вильнюса	Макс Фрай	480
5	Реверс	Сергей Лукьяненко, Александр Громов	352

Например, чтобы вывести все заказы пользователя «Александр Иванов» можно воспользоваться следующей SQL-конструкцией:

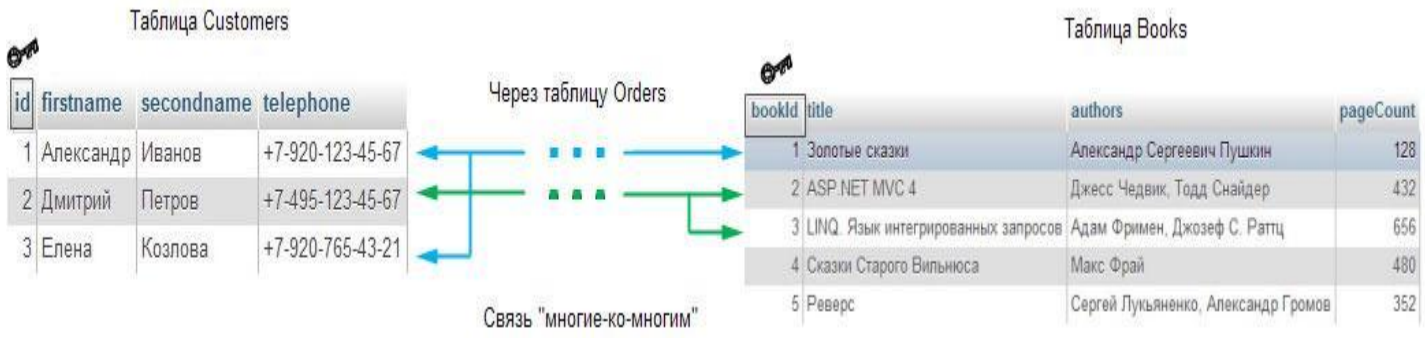
```
SELECT * FROM customers JOIN orders ON (customers.id = orders.userid) WHERE customers.id = 1;
```

+ Параметры

id	firstname	secondname	telephone	orderId	userId	bookId	dateOrder
1	Александр	Иванов	+7-920-123-45-67	1	1	1	2013-04-18 14:56:00
1	Александр	Иванов	+7-920-123-45-67	4	1	4	2013-05-02 14:12:00
1	Александр	Иванов	+7-920-123-45-67	5	1	5	2013-03-12 08:25:00

Связь «многие ко многим»

Связь «многие ко многим» возникает между двумя таблицами, когда в каждой из них может присутствовать несколько ключей другой таблицы. Например, покупатель приобретает в интернет-магазине сразу несколько книг. Или одну и ту же книгу заказывают несколько покупателей. На рисунке ниже показана связь «многие ко многим» между покупателями и приобретенными книгами:



Чтобы данные могли быть представлены в базе данных, этот тип связи преобразуется в две связи «один ко многим» с помощью **таблицы отображения (mapping table)**. В нашем случае такой таблицей является **Orders**.

Создание резервных копий и восстановление данных

Даже при грамотном администрировании баз данных иногда возникают определенные проблемы. Аппаратный сбой может привести к непредсказуемому поведению веб-страниц. Теперь, когда вы работаете с базой данных, простого резервного копирования файлов (HTML, PHP и изображений) на веб-сервере недостаточно. Нет ничего хуже, чем заставлять пользователей своего веб-сайта повторно вводить учетную информацию для регистрации. При наличии полной резервной копии вы сможете оценить разницу между восстановлением за час и повторным изобретением колеса. Рассмотрим несколько тактик резервного копирования баз данных.

Команда mysqldump

Гораздо лучше выполнять резервное копирование с помощью инструмента командной строки **MySQL**. Он позволяет создать резервную копию и восстановить данные, а также переместить БД с одного сервера на другой. Утилита **mysqldump** создает текстовый файл с инструкциями **SQL**, необходимыми для создания объектов БД и вставки данных.

Утилита **mysqldump** запускается из командной строки и принимает параметры для создания резервной копии единственной таблицы, базы данных и т.п. Синтаксис команды:

```
mysqldump -u пользователь -p пароль объекты_для_резервного_копирования
```

По умолчанию **mysqldump** создает и выводит резервную копию на стандартное устройство вывода (обычно это экран). Указанный пользователь должен иметь право на доступ к копируемым объектам. Перед копированием утилита предложит ввести пароль для данного пользователя. Чтобы выполнить копирование в файл, нужно добавить в конец команды символ (**>**) и указать имя файла.

Команды, выполняющие резервное копирование базы данных с именем **users** из командной строки:

```
mysqldump -u root -p users > my_backup.sql
```

Данная команда сообщает утилите **mysqldump** необходимость зарегистрироваться в базе данных с привилегиями пользователя **root** и создать резервную копию базы данных **users**. Перед копированием

у вас будет запрошен пароль пользователя **root**, указанный в процессе установки. Результат работы утилиты сохраняется в файле с именем **my_backup.sql** с помощью оператора перенаправления – символа «больше, чем» (**>**).

В этом сгенерированном коде **SQL** происходит создание базы данных **users** (если ее не существует) и генерация таблиц. Символы обратных апострофов (`), которыми окружены имена таблиц и столбцов в примере, использовать не обязательно.

Чтобы создать резервную копию единственной таблицы базы данных, достаточно просто добавить имя таблицы после имени БД. Например, следующая команда создает резервную копию таблицы **customers**:

```
mysqldump -u root -p users customers > my_backup_customers.sql
```

Но чаще всего вам понадобится создавать резервную копию всего содержимого базы данных. Это делается с помощью ключа командной строки ***--all-databases***. Результирующий файл содержит команды, необходимые для создания баз данных и пользователей, представляя собой полный снимок БД, пригодный для восстановления:

```
mysqldump -u root -p --all-databases > my_backup.sql
```

Пустая копия базы данных (только структура) создается с помощью ключа *--no-data*. Ключ *--no-create-info* позволяет выполнить противоположную операцию – создать только резервную копию данных. Разумеется, в резервной копии мало проку, если не знаешь, как восстановить из нее базу данных.

Восстановление из резервной копии

Восстановить базу данных из файла, созданного с помощью утилиты **mysqldump**, достаточно просто. Файл резервной копии – это просто набор инструкций SQL, которые могут исполняться клиентом командной строки **mysql** и восстанавливать данные из резервной копии.

Если резервная копия базы данных в файле **my_backup.sql** создавалась с ключом **--all-databases**, то восстановить базу данных можно так:

```
mysql -u root -p < my_backup.sql
```

Расширенный SQL

В этом разделе мы познакомимся с концепциями, которые, строго говоря, не являются необходимыми для создания веб-сайтов, но могут помочь повысить производительность и придать запросам большую гибкость.

Индексы

Индексы в базе данных играют ту же роль, что и алфавитный указатель в книге. Если вы попытаетесь найти в книге слова **CREATE TABLE** без алфавитного указателя, вам придется просмотреть значительное число страниц, чтобы обнаружить подходящий раздел, а потом полностью изучить его.

При таком способе очень неэффективно расходуется время – ваше или базы данных. Решение этой проблемы – добавить индексы.

Данные в индексах отсортированы и организованы таким образом, что позволяют находить требуемое значение настолько быстро, насколько это возможно. Поскольку значения отсортированы, база данных может прекратить поиск, обнаружив значение, превышающее искомое.

Но если индексы так хороши, почему бы не индексировать все подряд? Есть несколько причин:

- пространство, выделяемое под индексы, ограничено;
- даже в обычных книгах создание и обслуживание гигантских алфавитных указателей очень неэффективно;
- слишком большой объем данных в индексах приводит к увеличению времени их чтения при выборке данных.

Таким образом, решение о полях, включаемых в индексы, должно быть обоснованным. При исполнении простейшей инструкции **SELECT** (без инструкции **WHERE**) индексы не задействуются. Индексы используются в трех основных ситуациях:

1. **Вместе с оператором WHERE** – например, для выполнения запроса **SELECT * FROM customers WHERE firstname = 'Elena';** будет использован индекс по полю **firstname** (если он существует);
2. **Вместе с оператором ORDER BY** – например, для выполнения запроса **SELECT * FROM orders ORDER BY dateOrder;** будет использован индекс по полю **date Order** (если он существует);
3. Вместе с операторами **MIN** и **MAX** – например, для поля, передаваемого функции **MIN** или **MAX**, определен индекс.

Индексы должны быть определены до того, как они будут использоваться. Можно определить индексы для базы данных в команде **CREATE TABLE** или создать их позже для уже существующих таблиц с помощью специальных команд SQL. Если определение индекса является частью команды **CREATE TABLE**, оно указывается в конце блока кода, например так:

```
UNIQUE customers(firstname)
```

Команда **UNIQUE** создает индекс для поля с именем **firstname**. Тот же самый индекс можно создать с помощью специальной инструкции SQL:

```
CREATE UNIQUE INDEX authind ON Customers (firstname);
```

Попробуем получить описание таблицы **Customers**:

```
DESCRIBE Customers;
```

+ Параметры

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	*NULL*	auto_increment
firstname	varchar(32)	YES	UNI	*NULL*	
secondname	varchar(50)	YES		*NULL*	
telephone	varchar(20)	YES	UNI	*NULL*	

Обратите внимание на новое значение **UNI** в столбце **key** для поля **firstname**.

В MySQL можно создавать индексы, состоящие из нескольких столбцов (на приведенном выше рисунке уникальные индексы имеют столбцы **firstname** и **telephone**). Такие составные индексы обеспечивают уникальную комбинацию двух или больше полей. Лучшие кандидаты в индексы – это поля, которые с большой долей вероятности будут участвовать в конструкции **WHERE**. А если вы точно знаете, какие комбинации ключей будут использоваться, они более всего подходят для построения индексов, состоящих из нескольких столбцов.

Первыми в определении индекса должны следовать поля, которые используются наиболее часто. MySQL задействует составные индексы даже в том случае, если в запросе указано только первое значение, входящее в состав индекса. Уникальные индексы сродни первичному ключу, который также является уникальным. Но для каждой таблицы может быть определен только один первичный ключ. А уникальных индексов вы можете завести столько, сколько пожелаете.

Расширенная выборка данных

Мы уже рассматривали способ соединения таблиц в инструкции **SELECT** с помощью инструкции **WHERE**, но есть и другой способ. Если заменить ключевое слово **WHERE** словом **LEFT JOIN ON**, то будет выполнено левое, или *внешнее соединение (outer join)*. Левое соединение позволяет произвести запрос к двум таблицам, между которыми есть связь, но при этом для одной из таблиц возвращаются записи, даже если они не соответствуют записям в другой таблице. На примере наших таблиц можно было бы построить запрос, который возвращал бы список не только покупателей с их покупками, но и пользователей, которые не сделали ни одного заказа.

Синтаксис:

```
SELECT поля FROM левая_таблица
LEFT JOIN правая_таблица
ON левая_таблица.поле_связи = правая_таблица.поле_связи;
```

+ Параметры

title	Books
ASP.NET MVC 4	2
LINQ. Язык интегрированных запросов	1
Золотые сказки	2
Реверс	1
Сказки Старого Вильнюса	1

В таблице ниже приведен перечень функций, используемых с инструкцией **GROUP BY**:

Функции для работы со сгруппированными данными	
Функция	**Действие, выполняемое над сгруппированными данными**
COUNT()	Подсчет количества строк
SUM()	Подсчет суммы значений
AVG()	Вычисление среднего значения
MIN()	Определение минимального значения
MAX()	Определение максимального значения

Эти функции можно использовать в запросе и без инструкции **GROUP BY**. В этом случае все полученные данные рассматриваются как принадлежащие одной группе.

Функции базы данных

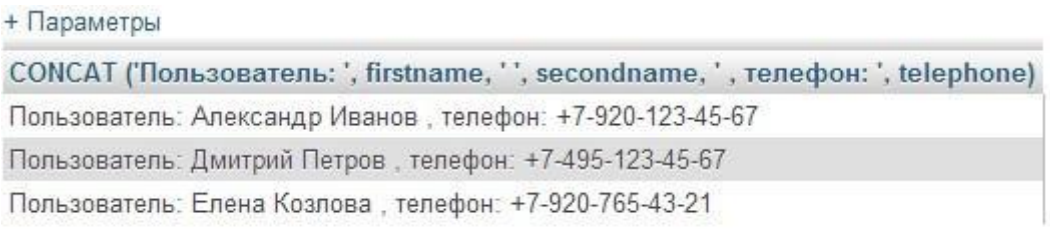
Как и в PHP-сценариях, в запросах к базе данных MySQL можно использовать функции. Рассмотрим несколько категорий, включая функции для работы со строками, а также с датой и временем.

Функции для работы со строками

Поскольку очень часто приходится работать со строками, MySQL предоставляет множество функций для решения разнообразных задач. Обычно строковые функции используются для работы с данными, возвращаемыми запросом. Но их можно задействовать, даже не ссылаясь на какие-либо таблицы.

В языке PHP объединение строк выполняется с помощью оператора точки (**.**). В MySQL есть аналогичная функция **CONCAT**, объединяющая строковые значения полей. Например, функция **CONCAT** позволяет вернуть единственное поле, соединяющее в себе имя покупателя и его номер телефона:

```
SELECT CONCAT ('Пользователь: ', firstname, ' ', secondname, ' , телефон: ',
telephone) FROM Customers;
```



Результатом конкатенации будет строка, готовая к отображению прямо из запроса SQL.

Имя поля, указываемое в качестве параметра функции, не нужно заключать в одинарные или двойные кавычки. Иначе MySQL примет его за литеральное значение. Функция CONCAT объединит столько полей, сколько вы ей зададите.

Иногда соединяемые при конкатенации поля разделяют некоторым символом. Это может потребоваться, например, при экспортировании таблиц. В таком случае следует применять функцию **CONCAT_WS**. Например, следующий запрос позволяет получить значения всех полей таблицы **Customers**, разделенные запятыми:

```
SELECT CONCAT_WS(',', *) FROM Customers;
```

В качестве символа-разделителя можно использовать пробел, что удобно для объединения имени и фамилии в единую строку, готовую к отображению.

Функции для работы с датой и временем

В языке PHP есть функции для работы с датой и временем, но как быть, если понадобилось запросить список покупок за последние 30 дней? Было бы замечательно иметь возможность выполнять арифметические операции над датой и временем прямо в запросах. В MySQL есть функции для подобной работы, применяемые как к значениям из таблиц базы данных, так и без упоминания таблиц в запросах. Продемонстрируем оба способа.

Дни, месяцы, годы и недели

Бывает трудно припомнить, глядя на дату, – вторник это был или четверг. В MySQL и в PHP есть функции, позволяющие мгновенно ответить на подобные вопросы.

Функция **WEEKDAY** принимает дату в качестве аргумента и возвращает число, означающее день недели: понедельнику соответствует 0, вторнику – 1 и т.д. Есть и аналогичная функция **DAYOFWEEK**, которая нумерует дни недели иначе – начиная с воскресенья, которому соответствует число 1. В примере ниже показано, как с помощью функции **WEEKDAY** определяется день недели, соответствующий 12 апреля 1961 года:

```
SELECT WEEKDAY('1961-04-12');
```

Этот запрос вернет число 2 – значит, 12 апреля 1961 года была среда. В MySQL есть и функция, которая возвращает название дня недели (на английском) – **DAYNAME**.

Есть еще функции **DAYOFMONTH** и **DAYOFYEAR**, аналогичные **DAYOFWEEK**. Они получают дату в качестве аргумента и возвращают число. Функция возвращает число месяца, а **DAYOFYEAR** –

количество дней, прошедших с начала календарного года. Есть функция **MONTHNAME**, аналогичная функции **DAYNAME**, которая возвращает название месяца.

Функция **WEEK** принимает в качестве аргумента дату и возвращает номер недели в году.

Часы, минуты и секунды

При работе с такими типами данных, как **datetime**, **timestamp** или **time**, в поле сохраняется указанное время. В MySQL есть несколько функций для работы с этим временем. Их имена вполне логичны: **HOUR**, **MINUTE** и **SECOND**. Функция **HOUR** принимает в качестве аргумента время и возвращает число часов в диапазоне от 0 до 23. Функция **MINUTE** возвращает число минут в диапазоне от 0 до 59, **SECOND** – число секунд в том же диапазоне.

MySQL предоставляет функции **DATE_ADD** и **DATE_SUB**, позволяющие складывать и вычитать даты. Их синтаксис:

```
DATE_ADD(дата, INTERVAL выражение тип)
DATE_SUB(дата, INTERVAL выражение тип)
```

Дата, которую вернет этот запрос, зависит от того, когда вы его выполните. Начиная с версии 3.23, MySQL поддерживает синтаксис операторов (+) и (-) для работы с датами:

```
-- Результат будет аналогичен предыдущему
SELECT NOW() - INTERVAL 12 day;
```

Функция **NOW** возвращает текущие дату и время согласно системным часам вашего компьютера (или сервера). Но если часы показывают неверную дату, то и функция **NOW** вернет ошибочную. В MySQL есть несколько функций, возвращающих текущую дату или время (либо текущую дату и время вместе). Функции **CURDATE** и **CURRENT_DATE** возвращают текущую дату в формате 'YYYY-MM-DD'. Функции **CURTIME** и **CURRENT_TIME** возвращают текущее время в формате 'HH:MM:SS'.

Транзакции

Транзакция – это механизм, который позволяет интерпретировать множественные изменения в базе данных как единую операцию. Либо будут приняты все изменения, либо все они будут отвергнуты. Ни из какого другого сеанса невозможно получить доступ к таблице, пока есть открытая транзакция, в рамках которой выполняются изменения в таблице. Если в своем сеансе вы попробуете сделать выборку данных сразу же после их изменения, все выполненные изменения будут доступны.

Такой механизм базы данных с поддержкой транзакций, как **InnoDB** или **BDB**, начинает транзакцию по команде **start transaction**, а завершает при подтверждении или отмене изменений. с помощью команды **commit** (сохраняет все изменения в базе данных) и **rollback** (отменяет все изменения).

В примере ниже создается таблица с поддержкой транзакций: в нее вставляются данные, затем запускается транзакция, в рамках которой данные удаляются, и в заключение выполняется откат транзакции (отмена удаления):

```
CREATE TABLE sample_innodb (
    id int(11) NOT NULL auto_increment,
    name varchar(150) default NULL,
PRIMARY KEY (id)
)
ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
INSERT INTO sample_innodb VALUES
      (1, 'Александр'),
      (2, 'Дмитрий');
```

```
start transaction;
DELETE FROM sample_innodb WHERE id = 1;
DELETE FROM sample_innodb WHERE id = 2;
rollback;
```

Поскольку произошел откат транзакции, данные из таблицы не были удалены.

Практическое задание

1. Подгрузка контента с помощью AJAX.
    a. На базе движка из курса «PHP. Уровень 1» взять модуль каталога.
    b. Выводить не все товары разом, а подгружать по 25 по нажатию кнопки «Еще».
2. * Создать очень много товаров и попробовать дойти до конца списка. Что происходит? Почему?

Дополнительные материалы

1. Сейед Тахагхогхи, Хью Вильямс. Руководство по MySQL.
2. http://devenergy.ru/archives/category/sql – работа с третьей нормальной формой и другие статьи по MySQL.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Мэтт Зандстра. PHP. Объекты, шаблоны и методики программирования.