69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01713 432003

Lecture Sheet on "PHP+MySQL"
Day-8: (MySQL 2)

Objectives	Topics
❖ Normalize a database into First, Second and Third Normal Forms ❖ Create and populate a database in MySQL ❖ Search a database using joins, groupings, and FULLTEXT searches ❖ Create indexes to improve the ability to search a database	❖ Normalization, keys, and relationships ❖ First Normal Form ❖ Second Normal Form ❖ Third Normal Form ❖ Creating and populating the database ❖ Performing joins ❖ Grouping selected results ❖ Creating indexes ❖ Using different table types ❖ Performing basic FULLTEXT searches ❖ Performing Boolean FULLTEXT searches ❖ Optimizing the database

SpreadSheetSyndrome: a tendency for the developer to lump every possible piece of information into as few table as possible, often into a single table. A schema that suffers from the ***Spreadsheet Syndrome*** is subject to data redundancies, data anomalies, and various inefficiencies. The cure for Spreadsheet Syndrome is database normalization.

Database normalization is a process by which an existing schema is modified to bring its component tables into compliance with a series of progressive normal forms. The concept of database normalization was first introduced by Edgar Frank Codd in his paper A Relational Model of Data for Large Shared Data Banks

The goal of database normalization is to ensure that every non-key column in every table is directly dependent ***on the key, the whole key and nothing but the key*** and with this goal come benefits in the form of reduced redundancies, fewer anomalies, and improved efficiencies. While normalization is not the be-all and end-all of good design, a normalized schema provides a good starting point for further development.

A Bookstore

Let's say you were looking to start an online bookstore. You would need to track certain information about the books available to your site viewers, such as:

- Title
- Author
- Author Biography
- ISBN
- Price
- Subject
- Number of Pages
- Publisher
- Publisher Address
- Description
- Review
- Reviewer Name

Let's start by adding the book that coined the term *Spreadsheet Syndrome*. Because this book has two authors, we are going to need to accommodate both in our table. Lets take a look at a typical approach

Table 1. Two Books

Title	Author	Bio	ISBN	Subject	Pages	Publisher
Beginning MySQL Database Design and Optimization	Chad Russell, Jon Stephens	Chad Russell is a programmer and network administrator who owns his own Internet hosting company., Jon Stephens is a member of the MySQL AB documentation team.	1590593324	MySQL, Database Design	520	Apress

Lets take a look at some issues involved in this design:

First, this table is subject to several anomalies: we cannot list publishers or authors without having a book because the ISBN is a primary key which cannot be NULL (referred to as an insertion anomaly). Similarly, we cannot delete a book without losing information on the authors and publisher (a deletion anomaly). Finally, when updating information, such as an author's name, we must change the data in every row, potentially corrupting data (an update anomaly).

Note: Normalization is a part of relational theory, which requires that each relation (AKA table) has a primary key. As a result, this article assumes that all tables have primary keys, without which a table cannot even be considered to be in first normal form.

Second, this table is not very efficient with storage. Lets imagine for a second that our publisher is extremely busy and managed to produce 5000 books for our database. Across 5000 rows we would need to store information such as a **publisher name, address, phone number, URL, contact email,** etc. All that information repeated over 5000 rows is a serious waste of storage resources.

Third, this design does not protect **data consistency**. Lets once again imagine that Jon Stephens has written 20 books. **Someone has had to type his name into the database 20 times**, and it is possible that his name will be misspelled at least once (i.e.. **John Stevens instead of Jon Stephens**). Our data is now in an inconsistent state, and anyone searching for a book by author name will find some of the results missing. This also contributes to the update anomalies mentioned earlier.

First Normal Form

The normalization process involves getting our data to conform to progressive normal forms, and a higher level of normalization cannot be achieved unless the previous levels have been satisfied (though many experienced designers can create normalized tables directly without iterating through the lower forms). The first normal form (or **1NF**) requires that the values in each column of a table are atomic. By atomic we mean that there are **no sets of values within a column**.

In our example table, we have a set of values in our author and subject columns. With more than one value in a single column, it is difficult to search for all books on a given subject or by a specific author. In addition, the author names themselves are non-atomic: first name and last name are in fact different values. Without separating first and last names it becomes difficult to sort on last name.

One method for bringing a table into first normal form is to **separate the entities contained in the table into separate tables**. In our case this would result in Book, Author, Subject and Publisher tables.

Table 2. Book Table

ISBN	Title	Pages
1590593324	Beginning MySQL Database Design and Optimization	520

Table 3. Author Table

Author_ID	First_Name	Last_name
1	Chad	Russell
2	Jon	Stephens
3	Mike	Hillyer

Table 4. Subject Table

Subject_ID	Name
1	MySQL
2	Database Design

Table 5. Publisher Table

Publisher_ID	Name	Address	City	State	Zip
1	Apress	2560 Ninth Street, Station 219	Berkeley	California	94710

Note The Author, Subject, and Publisher tables use what is known as a surrogate primary key -- an artificial primary key used when a natural primary key is either unavailable or impractical. In the case of author we cannot use the combination of first and last name as a primary key because there is no guarantee that each author's name will be unique, and we cannot assume to have the author's government ID number (such as SIN or SSN), so we use a surrogate key.

> Some developers use surrogate primary keys as a rule, others use them only in the absence of a natural candidate for the primary key. From a performance point of view, an integer used as a surrogate primary key can often provide better performance in a join than a **composite primary key** across several columns. However, when using a surrogate primary key it is still important to create a UNIQUE key to ensure that duplicate records are not created inadvertently (but some would argue that if you need a UNIQUE key it would be better to stick to a composite primary key).

By separating the data into different tables according to the entities each piece of data represents, we can now overcome some of the anomalies mentioned earlier: we can add authors who have not yet written books, we can delete books without losing author or publisher information, and information such as author names are only recoded once, preventing potential inconsistencies when updating.

Depending on your point of view, the Publisher table may or may not meet the 1NF requirements because of the Address column: on the one hand it represents a single address, on the other hand it is a concatenation of a building number, street number, and street name.

The decision on whether to further break down the address will depend on how you intend to use the data: if you need to query all publishers on a given street, you may want to have separate columns. If you only need the address for mailings, having a single address column should be acceptable (but keep potential future needs in mind).

Defining Relationships

As you can see, while our data is now split up, relationships between the tables have not been defined. There are various types of relationships that can exist between two tables:

*** One to (Zero or) One * One to (Zero or) Many * Many to Many**

The relationship between the Book table and the Author table is a many-to-many relationship: A book can have more than one author, and an author can write more than one book. To represent a many-to-many relationship in a relational database we need a third table to serve as a link between the two. By naming the table appropriately, it becomes instantly clear which tables it connects in a many-to-many relationship (in the following example, between the Book and the Author table).

Table 6. Book_Author Table

ISBN	Author_ID
1590593324	1
1590593324	2

Similarly, the Subject table also has a many-to-many relationship with the Book table, as a book can cover multiple subjects, and a subject can be explained by multiple books:

Table 7. Book_Subject Table

ISBN	Subject_ID
1590593324	1
1590593324	2

As you can see, we now have established the relationships between the Book, Author, and Subject tables. A book can have an unlimited number of authors, and can refer to an unlimited number of subjects. We can also easily search for books by a given author or referring to a given subject.

The case of a one-to-many relationship exists between the Book table and the Publisher table. A given book has only one publisher (for our purposes), and a publisher will publish many books. When we have a one-to-many relationship, we place a foreign key in the table representing the **many**, pointing to the primary key of the table representing the **one**. Here is the new Book table:

Table 8. Book Table

ISBN	Title	Pages	Publisher_ID
1590593324	Beginning MySQL Database Design and Optimization	520	1

Since the Book table represents the **many** portion of our one-to-many relationship, we have placed the primary key value of the Publisher as in aPublisher_ID column as a foreign key.

In the tables above the values stored refer to primary key values from the Book, Author, Subject and Publisher tables. Columns in a table that refer to primary keys from another table are known as foreign keys, and serve the purpose of defining data relationships.

In database systems (DBMS) which support referential integrity constraints, such as the InnoDB storage engine for MySQL, defining a column as a foreign key will allow the DBMS to enforce the relationships you define. For example, with foreign keys defined, the InnoDB storage engine will not allow you to insert a row into the Book_Subject table unless the book and subject in question already exist in the Book and Subject tables or if you're inserting NULL values. Such systems will also prevent the deletion of books from the book table that have child entries in the Book_Subject or Book_Author tables.

Second Normal Form

Where the First Normal Form deals with atomicity of data, the Second Normal Form (or 2NF) deals with relationships between composite key columns and non-key columns. As stated earlier, the normal forms are progressive, so to achieve Second Normal Form, your tables must already be in First Normal Form.

The second normal form (or **2NF**) **any non-key columns must depend on the entire primary key**. In the case of a composite primary key, this means that a non-key column cannot depend on only part of the composite key.

Let's introduce a Review table as an example.

Table 9. Review Table

ISBN	Author_ID	Summary	Author_URL
1590593324	3	A great book!	http://www.openwin.org

In this situation, the URL for the author of the review depends on the Author_ID, and not to the combination of Author_ID and ISBN, which form the composite primary key. To bring the Review table into compliance with 2NF, the Author_URL must be moved to the Author table.

Third Normal Form

Third Normal Form (**3NF**) requires that all columns depend directly on the primary key. Tables violate the Third Normal Form when one column depends on another column, which in turn depends on the primary key (a transitive dependency).

One way to identify transitive dependencies is to look at your table and **see if any columns would require updating if another column in the table was updated**. If such a column exists, it probably violates 3NF.

In the Publisher table the City and State fields are really dependent on the Zip column and not the Publisher_ID. To bring this table into compliance with Third Normal Form, we would need a table based on zip code:

Table 10. Zip Table

Zip	City	State
94710	Berkeley	California

In addition, you may wish to instead have separate City and State tables, with the City_ID in the Zip table and the State_ID in the City table.

A complete normalization of tables is desirable, but you may find that in practice that full normalization can introduce complexity to your design and application. More tables often means more JOIN operations, and in most database management systems (DBMSs) such JOIN operations can be costly, leading to decreased performance. The key lies in finding a balance where the first three normal forms are generally met without creating an exceedingly complicated schema.

Advantages of normalization

1. Smaller database: By eliminating duplicate data, you will be able to reduce the overall size of the database.
2. Better performance:
    a. Narrow tables: Having more fine-tuned tables allows your tables to have less columns and allows you to fit more records per data page.
    b. Fewer indexes per table mean faster maintenance tasks such as index rebuilds.
    c. Only join tables that you need.

Disadvantages of normalization

1. More tables to join: By spreading out your data into more tables, you increase the need to join tables.
2. Tables contain codes instead of real data: Repeated data is stored as codes rather than meaningful data. Therefore, there is always a need to go to the lookup table for the value.
3. Data model is difficult to query against: The data model is optimized for applications, not for **ad hoc querying**. It means that you almost cant query on a single table by looking at it.

Joining Tables

With our tables now separated by entity, we join the tables together in our SELECT queries and other statements to retrieve and manipulate related data. When joining tables, there are a variety of JOIN syntaxes available, but typically developers use the INNER JOIN and OUTER JOIN syntaxes.

An INNER JOIN query returns one row for each pair or matching rows in the tables being joined. Take our Author and Book_Author tables as an example:

```
mysql> SELECT First_Name, Last_Name, ISBN
    -> FROM Author INNER JOIN Book_Author ON Author.Author_ID =
Book_Author.Author_ID;
+------------+-----------+------------+
| First_Name | Last_Name | ISBN       |
+------------+-----------+------------+
| Chad       | Russell   | 1590593324 |
| Jon        | Stephens  | 1590593324 |
+------------+-----------+------------+
2 rows in set (0.05 sec)
```

The third author in the Author table is missing because there are no corresponding rows in the Book_Author table. When we need at least one row in the result set for every row in a given table, regardless of matching rows, we use an OUTER JOIN query.

There are three variations of the OUTER JOIN syntax: LEFT OUTER JOIN, RIGHT OUTER JOIN and FULL OUTER JOIN. The syntax used determines which table will be fully represented. A LEFT OUTER JOIN returns one row for each row in the table specified on the left side of the LEFT OUTER JOIN clause. The opposite is true for the RIGHT OUTER JOIN clause. A FULL OUTER JOIN returns one row for each row in both tables.

In each case, a row of NULL values is substituted when a matching row is not present. The following is an example of a LEFT OUTER JOIN:

```
mysql> SELECT First_Name, Last_Name, ISBN
    -> FROM Author LEFT OUTER JOIN Book_Author ON Author.Author_ID =
Book_Author.Author_ID;
+------------+-----------+------------+
| First_Name | Last_Name | ISBN       |
+------------+-----------+------------+
| Chad       | Russell   | 1590593324 |
| Jon        | Stephens  | 1590593324 |
| Mike       | Hillyer   | NULL       |
+------------+-----------+------------+
3 rows in set (0.00 sec)
```

The third author is returned in this example, with a NULL value for the ISBN column, indicating that there are no matching rows in the Book_Author table.

Resources

The following resources were either used in the development of this article or are considered to be of interest by the author.

- A Relational Model of Data for Large Shared Data Banks - E.F. Codd
- Database Normalization - Wikipedia
- A Simple Guide to Five Normal Forms in Relational Database Theory - William Kent
- Normal Form Definitions and Examples
- MySQL Database Design and Optimization - Jon Stephens & Chad Russell

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01713 432003

Self Joins

Suppose that a problem was found with a product (item id `DTNTR`), and you therefore wanted to know all of the products made by the same vendor so as to determine if the problem applied to them, too. This query requires that you first find out which vendor creates item `DTNTR`, and next find which other products are made by the same vendor. The following is one way to approach this problem:

• **Input** SELECT prod_id, prod_name FROM products WHERE vend_id = (SELECT vend_id FROM products WHERE prod_id = 'DTNTR');

• **Output**

```
+---------+----------------+
| prod_id | prod_name      |
+---------+----------------+
| DTNTR   | Detonator      |
| FB      | Bird seed      |
| FC      | Carrots        |
| SAFE    | Safe           |
| SLING   | Sling          |
| TNT1    | TNT (1 stick)  |
| TNT2    | TNT (5 sticks) |
+---------+----------------+
```

• **Analysis** This first solution uses subqueries. The inner `SELECT` statement does a simple retrieval to return the `vend_id` of the vendor that makes item `DTNTR`. That ID is the one used in the `WHERE` clause of the outer query so all items produced by that vendor are retrieved.

Now look at the same query using a join:

• **Input** SELECT p1.prod_id, p1.prod_name FROM products AS p1, products AS p2 WHERE p1.vend_id = p2.vend_id AND p2.prod_id = 'DTNTR';

• **Output**

```
+---------+----------------+
| prod_id | prod_name      |
+---------+----------------+
| DTNTR   | Detonator      |
| FB      | Bird seed      |
| FC      | Carrots        |
| SAFE    | Safe           |
| SLING   | Sling          |
| TNT1    | TNT (1 stick)  |
| TNT2    | TNT (5 sticks) |
+---------+----------------+
```

• **Analysis** The two tables needed in this query are actually the same table, and so the `products` table appears in the `FROM` clause twice. Although this is perfectly legal, any references to table `products` would be ambiguous because MySQL could not know to which instance of the `products` table you are referring.

To resolve this problem, table aliases are used. The first occurrence of `products` has an alias of `p1`, and the second has an alias of `p2`. Now those aliases can be used as table names. The `SELECT` statement, for example, uses the `p1` prefix to explicitly state the full name of the desired columns. If it did not, MySQL would return an error because there are two columns named `prod_id` and `prod_name`. It cannot know which one you want (even though, in truth, they are one and the same). The `WHERE` clause first joins the tables (by matching `vend_id` in `p1` to `vend_id` in `p2`), and then it filters the data by `prod_id` in the second table to return only the desired data.

Tip Self Joins Instead of Subqueries Self joins are often used to replace statements using subqueries that retrieve data from the same table as the outer statement. Although the end result is the same, sometimes these joins execute far more quickly than they do subqueries. It is usually worth experimenting with both to determine which performs better.

Natural Joins

Whenever tables are joined, at least one column appears in more than one table (the columns being joined). Standard joins (the inner joins you learned about in the previous tutorial) return all data, even multiple occurrences of the same column. **A natural join simply eliminates those multiple occurrences so only one of each column is returned.**

How does it do this? The answer is it doesn't; you do it. A natural join is a join in which you select only columns that are unique. This is typically done using a wildcard (`SELECT *`) for one table and explicit subsets of the columns for all other tables. The following is an example:

• Input

```
SELECT c.*, o.order_num, o.order_date,
       oi.prod_id, oi.quantity, OI.item_price
FROM customers AS c, orders AS o, orderitems AS oi
WHERE c.cust_id = o.cust_id
  AND oi.order_num = o.order_num
  AND prod_id = 'FB';
```

• **Analysis** In this example, a wildcard is used for the first table only. All other columns are explicitly listed so no duplicate columns are retrieved.

The truth is, every inner join you have created thus far is actually a natural join, and you will probably never even need an inner join that is not a natural join.

Outer Joins

Most joins relate rows in one table with rows in another. But occasionally, you want to include rows that have no related rows. For example, you might use joins to accomplish the following tasks:

- Count how many orders each customer placed, including customers who have yet to place an order
- List all products with order quantities, including products not ordered by anyone
- Calculate average sale sizes, taking into account customers who have not yet placed an order

In each of these examples, the join includes table rows that have no associated rows in the related table. This type of join is called an outer join.

The following `SELECT` statement is a simple inner join. It retrieves a list of all customers and their orders:

• **Input** SELECT customers.cust_id, orders.order_num FROM customers INNER JOIN orders ON customers.cust_id = orders.cust_id;

Outer join syntax is similar. To retrieve a list of all customers, including those who have placed no orders, you can do the following:

• **Input** SELECT customers.cust_id, orders.order_num FROM customers LEFT OUTER JOIN orders ON customers.cust_id = orders.cust_id;

• **Output**

```
+---------+-----------+
| cust_id | order_num |
+---------+-----------+
|   10001 |     20005 |
|   10001 |     20009 |
|   10002 |      NULL |
|   10003 |     20006 |
|   10004 |     20007 |
|   10005 |     20008 |
+---------+-----------+
```

• **Analysis** Like the inner join seen in the previous tutorial, this `SELECT` statement uses the keywords `OUTER JOIN` to specify the join type (instead of specifying it in the `WHERE` clause). But unlike inner joins, which relate rows in both tables, outer joins also include rows with no related rows. When using `OUTER JOIN` syntax you must use the `RIGHT` or `LEFT` keywords to specify the table from which to include all rows (`RIGHT` for the one on the right of `OUTER JOIN`, and `LEFT` for the one on the left). The previous example uses `LEFT OUTER JOIN` to select all the rows from the table on the left in the `FROM` clause (the `customers` table). To select all the rows from the table on the right, you use a `RIGHT OUTER JOIN` as seen in this example:

• **Input** SELECT customers.cust_id, orders.order_num FROM customers RIGHT OUTER JOIN orders ON orders.cust_id = customers.cust_id;

Note No `*=` MySQL does not support the use of the simplified `*=` and `=*` syntax popularized by other DBMSs.

Tip Outer Join Types There are two basic forms of outer joinsthe left outer join and the right outer join. The only difference between them is the order of the tables they are relating. In other words, a left outer join can be turned into a right outer join simply by reversing the order of the tables in the `FROM` or `WHERE` clause. As such, the two types of outer join can be used interchangeably, and the decision about which one is used is based purely on convenience.

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01713 432003

GROUP BY - Aggregate Functions

After you have mastered the basics of MySQL, it's time to take the next step and take on Aggregate Functions. Before we talk about what they are, let's review the definition of aggregate, as it relates to MySQL:

- **Aggregate** - Constituting or amounting to a whole; total. ~American Heritage Dictionary

With this type of wording, we can assume that MySQL's aggregate functions are something that will be very top-level, or in other words, the opposite of detailed.

The most common types of aggregate functions let you find out things like the minimum, maximum and even the average of a "grouped" set of data. The trick to understanding aggregate functions often understands what kind of data is being grouped and analyzed.

GROUP BY - The Data

Before we can start throwing around these fancy functions, let's build an appropriate table that has enough data in it to be meaningful to us. Below is the SQL for our "products" table. You can either run this SQL statement in your MySQL administrator software or use MySQL to execute the queries (i.e. create table, then each of the records).

Below is the MySQL table *products*.

Products Table:

id	name	type	price
123451	Park's Great Hits	Music	19.99
123452	Silly Puddy	Toy	3.99
123453	Playstation	Toy	89.95
123454	Men's T-Shirt	Clothing	32.50
123455	Blouse	Clothing	34.97
123456	Electronica 2002	Music	3.99
123457	Country Tunes	Music	21.55
123458	Watermelon	Food	8.73

GROUP BY - Creating Your First "Group"

Imagine that our store was running an advertisement in the newspaper and we wanted to have a "bargain basement" section that listed the lowest price of each product type. In this case we would be "grouping" by the product type and finding the minimum price of each group.

Our query needs to return two columns: product type and minimum price. Additionally, we want to use the *type* column as our group. The SELECT statement we are about to use will look different because it includes an aggregate function, MIN, and the GROUP BY statement, but otherwise it isn't any different than a normal SELECT statement.

PHP and MySQL Code:

```
<?php
// Make a MySQL Connection

$query = "SELECT type, MIN(price) FROM products GROUP BY type";

$result = mysql_query($query) or die(mysql_error());

// Print out result
while($row = mysql_fetch_array($result)){
        echo $row['type']. " - $". $row['MIN(price)'];
        echo "<br />";
}
?>
```

Our "products" table has four types of products: Music, Toy, Clothing and Food. When we GROUP BY *type* then we get one result for each of these types.

Display:

Clothing - $32.50
Food - $8.73
Music - $3.99
Toy - $3.99

MySQL GROUP BY - Review

Group BY is good for retrieving information about a group of data. If you only had one product of each type, then GROUP BY would not be all that useful.

GROUP BY only shines when you have many similar things. For example, if you have a number of products of the same type, and you want to find out some statistical information like the minimum, maximum, or other top-level info, you would use GROUP BY.

Some technical rules of GROUP BY:

- The column that you GROUP BY must also be in your SELECT statement.
- Remember to group by the column you want information about and **not the one you are applying the aggregate function on**. In our above example we wanted information on the *type* column and the aggregate function was applied to the *price* column.

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01713 432003

FullText Searches

```
CREATE TABLE articles (
     id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
     title VARCHAR(200),
     body TEXT,
     FULLTEXT (title,body) ←
   );

mysql> INSERT INTO articles (title,body) VALUES
   ('MySQL Tutorial','DBMS stands for DataBase ...'),
   ('How To Use MySQL Well','After you went through a ...'),
   ('Optimizing MySQL','In this tutorial we will show ...'),
   ('1001 MySQL Tricks','1. Never run mysqld as root. 2. ...'),
   ('MySQL vs. YourSQL','In the following database comparison ...'),
   ('MySQL Security','When configured properly, MySQL ...');

mysql> SELECT * FROM articles
    -> WHERE MATCH (title,body) AGAINST ('database');

mysql> SELECT id, MATCH (title,body) AGAINST ('Tutorial')
    -> FROM articles;
mysql> SELECT id, body, MATCH (title,body) AGAINST
    -> ('Security implications of running MySQL as root') AS score
    -> FROM articles WHERE MATCH (title,body) AGAINST
    -> ('Security implications of running MySQL as root');
```

Optimization of Database

From its inception, speed has been a strong point of the MySQL database server. In fact, its developers have long been cautious to add new features at the expense of performance even when faced by withering pressure from detractors. Yet over time MySQL's features caught up with its blazing speed, and today its used to power some of the highest traffic websites in the world, Yahoo! Finance, Craiglist, and TicketMaster among them. Accordingly, for most applications chances are MySQL is going to perform to your expectations.

However, with Web traffic exploding, and the complexity of applications increasing all the time, you should strive to optimize your database from its very inception, and continue to review and refine its structure and query activity over its lifetime. You can do so by following a process I refer to as NICM, or preferably, "Neglecting to Index Causes Misery" (those of you new to database optimization will soon get the joke). This process consists of five steps, including *Normalize*, *Index*, *Cache*, and *Monitor*. In this inaugural installment of my new MySQL series for Developer.com, I'll introduce these five steps, highlighting not only how they will ensure your database is running at full tilt, but also how they will actually help you to more effectively manage your data.

Normalize

The relational database is aptly named because it promotes the strategy of managing data through well-defined relations. By creating and enforcing relations, its possible to greatly reduce the possibility inconsistencies could creep into the data. This strategy is known as *database normalization*, of which there are several well-defined states, also known as *forms*.

To illustrate both the concept and importance of normalization, consider a scenario in which you're tasked with creating a corporate human resources application capable of managing employee information. This application would allow the HR manager to easily insert, update, view and delete employee data. Each employee record would contain the usual information, including name, phone number, position, salary, and supervisor.

Note that in the opening sentence I stated that relational databases *promote* the strategy of managing data through well-defined relations—you're certainly not constrained to do so. Therefore there's nothing to prevent you from creating a single table that looks like this:

```
CREATE TABLE employees (
 name VARCHAR(50) NOT NULL,
 telephone CHAR(10) NOT NULL,
 position VARCHAR(20) NOT NULL,
 salary DECIMAL(8,2) NOT NULL,
 supervisor VARCHAR(50) NOT NULL
)
```

Yet several problems with this approach should be evident. Because the data types used to define these columns are all indicative of an input method requiring the manager to manually enter each value. For example, the manager might type the following values into the web form when adding a new employee:

"Jason Gilmore", "614-999-9999", "Software Developer", "50000.00", "John Thompson"

But what if over time the manager begins to enter the "Software Developer" title as "Sft. Dev.", and occasionally forgets the periods, using just "Sft Dev"? These inconsistencies eliminate the possibility of using simple SELECT queries to retrieve the roster of software developers. The simple answer to the problem is to normalize the position data, creating a new table intended to contain the master list of all possible positions:

```
CREATE TABLE positions (
 id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 name VARCHAR(20) NOT NULL
)
```

Then you revise the employees table so that it refers to a position's primary key rather than the name (in the revised table, the _fk postfix is simply a naming convention to remind the database administrator that this column points to a foreign table key):

```
CREATE TABLE employees (
 name VARCHAR(50) NOT NULL,
 telephone CHAR(10) NOT NULL,
 position_fk TINYINT UNSIGNED NOT NULL,
 salary DECIMAL(8,2) NOT NULL,
 supervisor VARCHAR(50) NOT NULL
);
```

Now you can retrieve the positions from the positions table and use them to populate a drop-down list. This forces the manager to choose from a constrained set rather than haphazardly type in the titles.

A similar normalization improvement can be made to the supervisor column, although this is a case where an additional table isn't necessary. All you need to do is modify the employees table anew so it looks like this:

```
CREATE TABLE employees (
 id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 name VARCHAR(50) NOT NULL,
 telephone CHAR(10) NOT NULL,
 position_fk TINYINT UNSIGNED NOT NULL,
 salary DECIMAL(8,2) NOT NULL,
 supervisor_pk SMALLINT UNSIGNED
);
```

Can you figure out the origin of the data used to populate the supervisor_pk column? That's right, the very same employees table!

I also suggest downloading MySQL's free MySQL Query Browser application, which can help you to visualize the data and its various relationships in a manner much more efficient than if you were using the command-line client.

Index

Once the tables have been properly designed and normalized, you should next take some time to think about what data will most commonly be queried, and create special data structures known as *indexes* which will dramatically improve the performance of these query operations. Indexes are important because they organize the indexed data in a way that allows MySQL to retrieve the desired record in the fastest possible fashion. In the previous section you already encountered an index, although it isn't apparent: the primary key. But the primary key will only speed query operations when you're searching for a record by using that primary key as the identifier, for example: **SELECT name, telephone FROM employees WHERE id='3'**

What if you wanted to search for one or several records based on the employee name? The query might look like this: **SELECT id FROM employees WHERE name='John Smith'**

To speed these sorts of queries, you'll need to add an index to the name column. The revised employees table looks like this:

```
CREATE TABLE employees (
 id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 name VARCHAR(50) NOT NULL,
 INDEX (name),
 telephone CHAR(10) NOT NULL,
 position_fk TINYINT UNSIGNED NOT NULL,
 salary DECIMAL(8,2) NOT NULL,
 supervisor_pk SMALLINT UNSIGNED
);
```

You can also create multiple-column indexes that would greatly improve the performance of queries such as this: **SELECT id FROM employees WHERE firstname = 'John' and lastname = 'Smith'**

The revised employees table would look like this (I've also broken the name into two separate columns as indicated by the query):

```
CREATE TABLE employees (
 id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
 firstname VARCHAR(20) NOT NULL,
 lastname VARCHAR(20) NOT NULL,
 INDEX name (firstname, lastname),
 telephone CHAR(10) NOT NULL,
 position_fk TINYINT UNSIGNED NOT NULL,
 salary DECIMAL(8,2) NOT NULL,
 supervisor_pk SMALLINT UNSIGNED
);
```

This new index would actually improve quite a few queries, including the following:

```
SELECT telephone, position FROM employees WHERE firstname = 'John'
SELECT * FROM employees WHERE firstname = 'John' and lastname = 'Smith'
```

Table indexing can be a rather confusing topic, although the time invested in learning its nuances will be well worth it. I suggest reading the following MySQL manual pages for more information:

Column Indexes Multiple-Column Indexes How MySQL Uses Indexes

Cache

MySQL is often used to dynamically generate web pages based on similar queries and rarely changing data. For instance, consider the home page of Developer.com, which likely depends on a single query used to repeatedly retrieve the latest ten or so articles. New articles are typically published early to mid-morning, meaning the relevant data found on this page really only chances perhaps once or twice a day. Why not cache the data returned by these queries, thereby bypassing the need to repeatedly retrieve it from the database? Introduced in version 4, enabling MySQL's query caching mechanism can result in a huge performance gain over neglecting to do so.

To learn more about how your MySQL installation's query caching mechanism is configured, run the following command:

```
mysql>SHOW VARIABLES LIKE '%query_cache%'
```

For more information about MySQL's caching feature, see 5.13. The MySQL Query Cache. Also see Jupitermedia sister site Database Journal for a great tutorial on the topic, authored by Ian Gilfillan.

Monitor

The reasoning behind this step should be obvious: how are you going to know what should be optimized if you're not actively monitoring how MySQL is operating? Thankfully, MySQL's developers have been particularly proactive in providing developers with the tools for keeping abreast of database performance.

For starters, you should familiarize yourself with MySQL's EXPLAIN query, which will provide detailed information regarding how MySQL goes about executing a SELECT query. You'll learn valuable tips regarding how MySQL is executing table joins, and whether additional indexes should be added.

If you're running a MySQL version earlier than 5.0, check out mytop, a console-based utility for monitoring MySQL performance.

Finally, be sure to have a solid understanding of MySQL's configuration variables, because they control crucial performance-related matters such as how much memory is allocated to queries, what sort of data is logged, how many simultaneous collections are allowed, and much more. You can review a complete list of these variables by executing:

mysql>SHOW VARIABLES;

I hope this introductory article left you thinking about how you can go about making your MySQL database absolutely scream. And moving forward, every time you begin a new database project, don't forget to approach it the NICM way!