

You are here: [Home](#) > [Developers](#) > [View Source Articles](#) > Java View Source Article

- Developer Program
 - [Membership](#)
 - [One-to-One Support](#)
 - [Newsgroups](#)
- Developer Publications
 - [View Source](#)
 - [Developer News](#)
- Documentation
 - [Technical Manuals](#)
 - [White Papers](#)
 - [TechNotes](#)
 - [Sample Code](#)
 - [FAQs](#)
 - [Books](#)
- Technologies
 - [CORBA](#)
 - [Directory & LDAP](#)
 - [Dynamic HTML](#)
 - [Java](#)
 - [JavaScript](#)
 - [Linux](#)
 - [Security](#)
 - [SSJS](#)
 - [XML](#)
- Developer Downloads
 - [Tools & SDKs](#)
 - [Patches](#)
- iPlanet Products
 - [Technical Resources](#)

The Nuts and Bolts of Relational Databases

A Primer for Web Developers

By [Duane K. Fields](#)

[Send comments and questions](#) about this article to [View Source](#).

Relational databases have long been a staple of big, well-funded enterprises, but lately they've been finding their way to a different audience of users. Nowadays, most large-scale web sites employ databases for at least some portion of their content. User registration information, inventory records, and event calendars are all commonly managed through a database. Consequently, some web developers are finding themselves faced with the challenge of getting up to speed on database development. If you've never developed database code and aren't sure how a database works, this article will help, by providing you with an introduction to the core concepts. Next month, I'll introduce JDBC - Java's API for database access.

Databases are simply applications, usually running on the server, that provide quick and efficient access to large amounts of organized data. In addition to storing this information, databases can establish a relationship among various pieces of data, providing powerful sorting, searching, and merging capabilities. Databases can also help manage access to data by supporting multiple users with different capabilities and levels of access.

DATABASE TABLES AND SCHEMA

The data inside a database is stored in tables, two-dimensional data structures that organize the information. A table is similar to a spreadsheet in that it's composed of a fixed number of columns and a variable number of rows, and in that the rows and columns intersect to form cells, each of which holds a piece of data. Each row of a table represents an entry; each column, identified by a specific name and type of data, holds a detail of an entry. It might help to think of a table row as a record and the columns as the fields. For example, if your Rolodex were turned into a table in a database, each card would become a row in the table while each field on the card (name, address, phone, fax) would become a column in the table.

Figure 1 shows what a table designed to hold information for a product catalog might look like if you peeked into the database and examined it. Each row represents a different inventory item, while details about that item are stored in the appropriate column. If we decide to add a new item to our product catalog, we simply insert another row of information into the table.

Figure 1. A product catalog table in the database

| ITEM_ID | DESCRIPTION | PRICE | QTY_AVAILABLE |
|---------|---------------------|--------|---------------|
| 100 | Yellow grid bug | 12.50 | 82 |
| 200 | Blue grid bug | 13.50 | 2 |
| 300 | Red grid bug | 9.95 | 165 |
| 400 | Light cycle battery | 4.87 | 112 |
| 500 | Energy pod | 112.99 | 25 |

Databases are typically composed of many tables, each of which is designed to handle a specific type of information and assigned a unique name by which you can reference its data. For example, the table above might be named

`PRODUCTS_TABLE`; the database that it's part of might have other tables as well - one to store registered user information, one to hold sales records, and so on. As you'll learn, one of the most powerful features of a database is its ability to combine data from a collection of interrelated tables into a single set of information.

Again, it can help to think of tables as spreadsheets - spreadsheets that you can programmatically access, sorting their columns and controlling their contents. Unlike in a spreadsheet, however, the columns contain only data - not formulas or calculations - and each column can hold only one specific type of data, such as a string, an integer, or a floating-point number. A table's collection of column names and data types is known as its schema. Whoever sets up the database also designs the schema to model the real-world data to be stored.

OBJECTS, JAVABEANS, AND DATABASE TABLES COMPARED

The rows of a table in a relational database have some similarity to simple Java objects or JavaBeans components. Where objects have instance variables and Beans have properties, tables have columns. A table's schema is like the class that defines a Bean, specifying the names and types of data that instances will hold. Each row of the table is an instance of whatever it is that the table represents, just as an object or Bean is an instance of its class. When building your applications, you may find it convenient to think of tables as a way to flatten out and store such objects or components for later use.

Like Java classes, tables are templates for storing a specific set of information like the data from a purchase order or details about inventory items, and they aren't particularly useful by themselves. It's only when we create instances of a Java class or add rows to a table that we have something of value. Both classes and tables, then, serve as a useful containers for managing information about some real-world object or event.

QUERYING THE DATABASE WITH SQL

To get data from a database, we issue queries in Structured Query Language (SQL). SQL is a simple language that specifies the syntax for constructing a search request and controlling how the results are returned. While there are some variations in syntax and capabilities among database vendors, the basic commands are generally compatible among databases.

An SQL query specifies what information we're interested in and what table it can be found in. In Java applications, this query is sent across the network to the database through Java's database API, either directly (in a scriptlet or expression) or via a database access Bean.

SQL queries generally return data in a tabular format, containing both rows and columns - just like the tables that store the original data. What we're left with is essentially a table of object instances and their properties. The table resulting from an SQL query doesn't necessarily have to correspond exactly to the structure of an existing table, however. The results of an SQL query might instead be only a subset, or even a modified collection of data.

Retrieving Data With SELECT Commands

One of the most fundamental SQL commands is

`SELECT`, which retrieves a set of information from a table. The syntax of the basic `SELECT` command is as follows:

```
SELECT column(s) FROM table(s)
```

(Note that although more than one table can be named, in practice you probably wouldn't want to do this unless you also included a

`WHERE` clause, as explained below in "Table Joins.") Here's a simple SQL query that asks the database to return all of the data stored in the products table we defined earlier:

```
SELECT * FROM PRODUCTS_TABLE
```

The "

"*" character means to include results from every column in the table. But we don't have to retrieve items from every column; we can also specify a subset of columns to return and say how the rows should be ordered. For example, consider the following query:

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS_TABLE ORDER BY PRICE
```

This query specifies that we want the results to include only the

DESCRIPTION and PRICE columns of the products table, in that order. It further specifies that we would like the rows to be ordered on the contents of the PRICE column. This query would return a results table like that shown in Figure 2.

Figure 2. Specified columns from products table, ordered by price

| DESCRIPTION | PRICE |
|---------------------|--------|
| Light cycle battery | 4.87 |
| Red grid bug | 9.95 |
| Yellow grid bug | 12.50 |
| Blue grid bug | 13.50 |
| Energy pod | 112.99 |

Applying Conditional Tests With the WHERE Clause

We can further limit our search results by applying one of a number of possible conditional tests to each row in the results table. A conditional test is specified by appending a

WHERE clause to the end of the SELECT command:

```
SELECT column(s) FROM table(s) WHERE condition
```

We can, for instance, select a single record from the table by constructing a

WHERE clause that looks for a particular value in a particular field. In our case, because each value in the ITEM_ID column in our products table is unique, we can retrieve a particular record simply by referencing this value. For example:

```
SELECT * FROM PRODUCTS_TABLE WHERE ITEM_ID = 200
```

The above query would return the table of results shown in Figure 3.

Figure 3. The record for item 200

| ITEM_ID | DESCRIPTION | PRICE | QTY_AVAILABLE |
|---------|---------------|-------|---------------|
| 200 | Blue grid bug | 13.50 | 2 |

We can also perform comparison operations against column values. For example, we can construct a query that returns only the records for items priced higher than \$10, as indicated by the value in each record's

PRICE column.

```
SELECT * FROM PRODUCTS_TABLE WHERE PRICE > 10
```

This query would return the table of results shown in Figure 4.

Figure 4. The records for items priced higher than \$10

| ITEM_ID | DESCRIPTION | PRICE | QTY_AVAILABLE |
|---------|-----------------|--------|---------------|
| 100 | Yellow grid bug | 12.50 | 82 |
| 200 | Blue grid bug | 13.50 | 2 |
| 500 | Energy pod | 112.99 | 25 |

We can perform a variety of other conditional tests as well, including equality and inequality, greater than or less than, and various string-matching tests. One useful test that we can perform on string values employs the

`LIKE` clause plus the wildcard character (%) to do a wildcard search. For instance, if we wanted to find only records whose `DESCRIPTION` column value ends in " grid bug", we could use this query:

```
SELECT * FROM PRODUCTS_TABLE WHERE DESCRIPTION LIKE '% grid bug'
```

Note that SQL requires that we enclose string constants in single quotation marks (double quotation marks won't work for most databases). We'll also need to escape special characters and follow strict date format conventions. In next month's discussion of the JDBC API, I'll show you some techniques for easing the burden that these restrictions place on the developer.

The SQL query above would produce the result set shown in Figure 5.

Figure 5. The records whose description ends in " grid bug"

| ITEM_ID | DESCRIPTION | PRICE | QTY_AVAILABLE |
|---------|-----------------|-------|---------------|
| 100 | Yellow grid bug | 12.50 | 82 |
| 200 | Blue grid bug | 13.50 | 2 |
| 300 | Red grid bug | 9.95 | 165 |

Many other tests, filters, and comparison operators can be included in SQL queries. One particularly interesting filter available to many systems, called

`SOUNDEX`, allows us to search for something that sounds like another word when pronounced phonetically. This means that we can construct "fuzzy" searches where the exact search term isn't known - for example, we could search on "Jennifer" and receive matches on "Jennifer", "Genifer", and "Jenifer".

The terms of a

`WHERE` clause aren't limited to a single test. We can combine a series of tests with Boolean `AND/OR` operators to further refine our search. Consider, for instance, the following query, which could be used to determine which grid bug varieties are running low on inventory:

```
SELECT DESCRIPTION, QTY_AVAILABLE FROM PRODUCTS_TABLE WHERE DESCRIPTION LIKE '% grid bug'
AND QTY_AVAILABLE < 30 ORDER BY QTY_AVAILABLE
```

Here we're getting only two columns from the products table and then limiting the rows returned to those that contain " grid bug". We're then further restricting the results to those records whose

`QTY_AVAILABLE` column indicates that there are fewer than 30 units left in inventory. These restrictions are cumulative, as specified by the `AND` operator in the query. Finally, we're ordering the rows based on the values in the `QTY_AVAILABLE` column. This search would return the results table shown in Figure 6.

Figure 6. Specified columns from the records for grid bugs with fewer than 30 left in inventory, ordered by quantity available

| DESCRIPTION | QTY_AVAILABLE |
|-------------|---------------|
|-------------|---------------|

| | |
|---------------|----|
| Blue grid bug | 2 |
| Energy pod | 25 |

It's important to note that we can perform conditional tests and sorting on the values of any column in the table, even if we don't plan to include that column in our results. For example, in the above search we could have ordered the values by

ITEM_ID, even though we chose not to return that column in the results.

TABLE RELATIONSHIPS

As mentioned earlier in this article, one of the key benefits of a relational database is its ability to establish relationships among the various tables being used to store information. Thus, we can store our data in separate but interrelated tables instead of trying to cram everything into one big table. For example, we could keep a record of all purchase orders we received in a table called

PO_TABLE, related by key values to our products table.

Key Values

To establish relationships among tables, we need to identify the key values in each table. Key values are values in a column that can be used as a reference to a particular record. For instance, the key to records in our products table is the

ITEM_ID column, which contains a unique part number for each product. In place of product descriptions, which are unwieldy and may change over time, the item ID gives us a stable, easy-to-manage way to refer to any item in our catalog. Once we have an item ID - anywhere, even inside another table - we can use it to pull out all of the other details about that product, such as its description or price.

The use of a unique key in each table can also make database access faster and more efficient. When the database expects queries based on the contents of a particular column, it can optimize its internal index files appropriately. Generally, we use integer values for keys because they allow for the greatest amount of optimization inside the database. When we use unique keys like this, the database can make further optimizations toward searching.

Let's now look at how two tables in a database can be related by key values. The table shown in Figure 7, called

PO_TABLE, stores data about pending purchase orders that need to be filled. Each record specifies the purchase order number, the item number, and the quantity ordered. The value in the ITEM column in this table corresponds to a value in the ITEM_ID column in our products table, thus establishing a relationship between the two tables. The ITEM column in this case is known as a foreign key, because it holds a value that corresponds to a key value in another table.

Figure 7. PO_TABLE

| PO_NUMBER | ITEM | QTY_PURCHASED |
|-----------|------|---------------|
| 8759362 | 100 | 5 |
| 8759362 | 200 | 1800 |
| 0987857 | 500 | 6 |
| 9913463 | 400 | 9 |
| 9913463 | 200 | 17 |

Table Joins

To merge information from two different tables in our table of results, we must name both tables in our SQL query. When a query combines data from more than one table, it's known as a join. A join names columns from several tables and typically uses a

WHERE clause to establish some relationship among the columns. If we don't specify a WHERE clause and just specify several tables and columns in our query, the database engine will return every possible combination, which is a big mess and probably not what we were intending! Consider the following query, which simply fetches an appropriate description from the products table for each item referenced in the purchase orders table:

```
SELECT PO_NUMBER, DESCRIPTION, QTY_PURCHASED FROM PO_TABLE, PRODUCTS_TABLE
WHERE ITEM = ITEM_ID
```

The logic behind the query goes something like this: We're requesting information that must be obtained from both the purchase orders table and the products table. We want the

PO_NUMBER and QTY_PURCHASED columns from the purchase orders table and the DESCRIPTION column from the products table. However, what we want to end up with is the description associated with each item listed in our purchase order table's ITEM field. We use a WHERE clause to select only that one particular intersection between the two tables. The results table that we get back from such a query looks like the one in Figure 8.

Figure 8. The result of our table join

| PO_NUMBER | DESCRIPTION | QTY_PURCHASED |
|-----------|---------------------|---------------|
| 8759362 | Blue grid bug | 5 |
| 8759362 | Yellow grid bug | 1800 |
| 0987857 | Energy pod | 6 |
| 9913463 | Light cycle battery | 9 |
| 9913463 | Yellow grid bug | 17 |

If your query includes tables that have columns with the exact same names, the database won't be able to determine which table's column you really want. To let it know, you must prefix the ambiguous column name with the name of its table (followed by a period). For example, if both our tables had a

DATE column, we would refer to them as PRODUCTS_TABLE.DATE and PO_TABLE.DATE.

It's also possible to perform more complex comparisons between columns of related tables. Here's an example of a query that will generate a table of purchase orders that will require back orders to be made, because the quantity requested in the purchase order is greater than the number of items currently in inventory:

```
SELECT PO_NUMBER FROM PO_TABLE, PRODUCTS_TABLE WHERE PO_TABLE.ITEM = PRODUCTS_TABLE.ITEM_ID
AND QTY_AVAILABLE < QTY_PURCHASED
```

MODIFYING DATABASE CONTENTS

SQL can do more than simply retrieve information already stored in the database, of course. We can also use SQL commands to create new records or update existing ones. The syntax is a little different from the

SELECT queries we've been studying but still pretty simple:

```
INSERT INTO table (column(s)) VALUES (value(s))
DELETE FROM table WHERE condition
UPDATE table SET column=value, column=value, ...
```

It's important to understand that these commands operate only on the values of columns in a row and not on columns in and of themselves. In other words, we don't delete or add columns themselves; we only delete or add rows or update the values of columns in a row.

INSERT Commands

INSERT, as you might expect, enables us to insert data into a table by adding rows. The INSERT command requires a list of columns to insert and a corresponding list of values. The first value listed is assigned to the first column listed, the second to the second, and so on. We can leave out the columns list, in which case the values are mapped to the table's columns in the order they were specified when we defined the table's schema (the "natural order" of the columns).

Here's an example of adding a new item to our products table:

```
INSERT INTO PRODUCTS_TABLE (ITEM_ID, DESCRIPTION) VALUES (600, 'Tan grid bug')
```

Figure 9 shows our table with this new row added. We didn't specify values for the

PRICE and QTY_AVAILABLE columns in our INSERT command, so these columns are still undefined.

Figure 9. The result of our insertion

| ITEM_ID | DESCRIPTION | PRICE | QTY_AVAILABLE |
|---------|---------------------|--------|---------------|
| 100 | Yellow grid bug | 12.50 | 82 |
| 200 | Blue grid bug | 13.50 | 2 |
| 300 | Red grid bug | 9.95 | 165 |
| 400 | Light cycle battery | 4.87 | 112 |
| 500 | Energy pod | 112.99 | 25 |
| 600 | Tan grid bug | | |

UPDATE Commands

We can add the missing values to our new row with an

UPDATE command, targeted at the record for the tan grid bug, which has an ITEM_ID value of 600:

```
UPDATE PRODUCTS_TABLE SET PRICE=19.95, QTY_AVAILABLE=100 WHERE ITEM_ID = 600
```

The

WHERE clause in the UPDATE command restricts the update to affect only the single row we want. Figure 10 shows what our table would look like after this query was executed.

Figure 10. The result of our update

| ITEM_ID | DESCRIPTION | PRICE | QTY_AVAILABLE |
|---------|---------------------|--------|---------------|
| 100 | Yellow grid bug | 12.50 | 82 |
| 200 | Blue grid bug | 13.50 | 2 |
| 300 | Red grid bug | 9.95 | 165 |
| 400 | Light cycle battery | 4.87 | 112 |
| 500 | Energy pod | 112.99 | 25 |
| 600 | Tan grid bug | 19.95 | 100 |

It's also possible to set values of a column in multiple rows at the same time. If we had a terrible fire at the warehouse that destroyed our entire inventory, we could issue this command, which would set the

QTY_AVAILABLE column for all our products to a value of 0:

```
UPDATE PRODUCTS_TABLE SET QTY_AVAILABLE=0;
```

This command could also be issued in response to a complete sellout of our inventory - certainly a more positive example!

DELETE Commands

The

DELETE command is used to remove rows from a table that match a specified condition. Its syntax is similar to that of the SELECT command, and it works pretty much the same way. For example, if we were to discontinue our line of red grid bugs, we would issue the following command to remove its entry from the products table:

```
DELETE FROM PRODUCTS_TABLE WHERE ID = 300
```

After this command was executed, all rows fitting the criteria specified in the

WHERE clause would be removed from the table.

OFF TO A GOOD START

There's certainly lots more to know about relational databases and relational database management systems (RDBMS), but what you've learned in this article can get you started. The database principles I've discussed here can be applied in many different languages, including Java, Perl, C, C++, and PHP3. Each language has its own particular API or tags for database access.

Learning more about how to add database support to your applications is well worth the effort. In even the simplest program design, the use of a database eliminates the need for you to create precarious data storage files and complex data structures. Manage data in a database, not in your program!

FURTHER RESOURCES

- Rafe Colburn, [Special Edition Using SQL](#)

View Source wants your feedback!
[Write to us](#) and let us know
what you think of this article.

[Duane K. Fields](#) is a senior engineer for the E-Business Enablement group of IBM's Tivoli Systems, where he creates web-based applications with Java and JSP. He lives in Austin, Texas. His first book, *Web Development with JavaServer Pages*, which he coauthored with Mark Kolb, will be available in February 2000.

(12.99)

Any sample code included above is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#)