Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 14

Working with Databases

In this chapter you will learn . . .

- The role that databases play in web development
- What are the most common commands in SQL
- How to access SQL databases in PHP
- How NoSQL database systems work
- How to work with NoSQL databases using Node
- What is GraphQL

Databases and Web Development

In this book, the relational DBMS used will be either **SQLite** or **MySQL** (or MariaDB) There are many other open-source and proprietary relational DBMS alternates to **MySQL**, such as **PostgreSQL, Oracle Database**, **IBM DB2**, and **Microsoft SQL Server**.

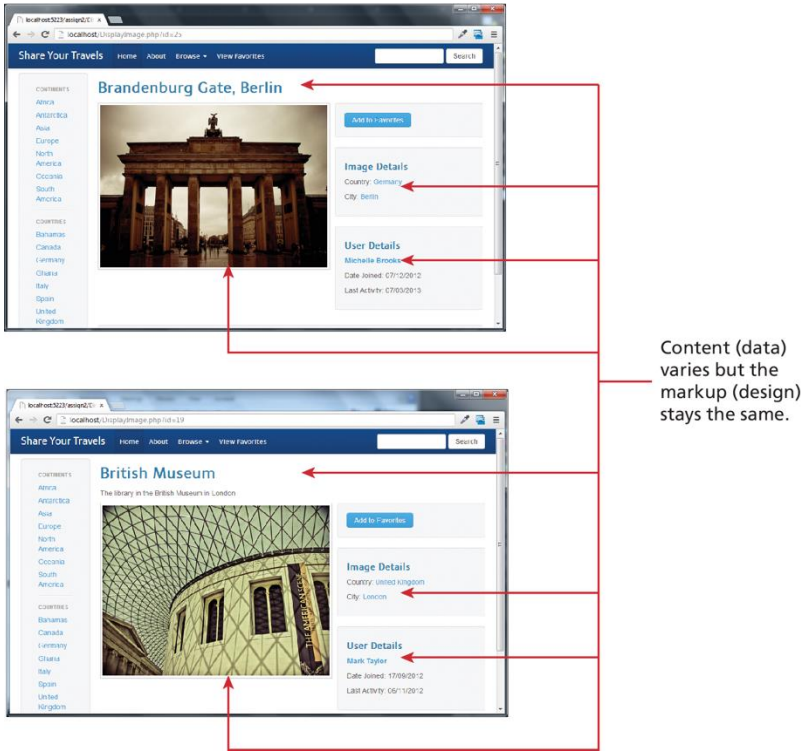
In addition to relational database systems, there are non-relational models for database systems that will also be explored in this chapter. These systems are usually categorized with the term **NoSQL** and includes systems such as **Cassandra** and **MongoDB**

Databases provide **data integrity** (accuracy and consistency of data) and can reduce the amount of **data duplication**

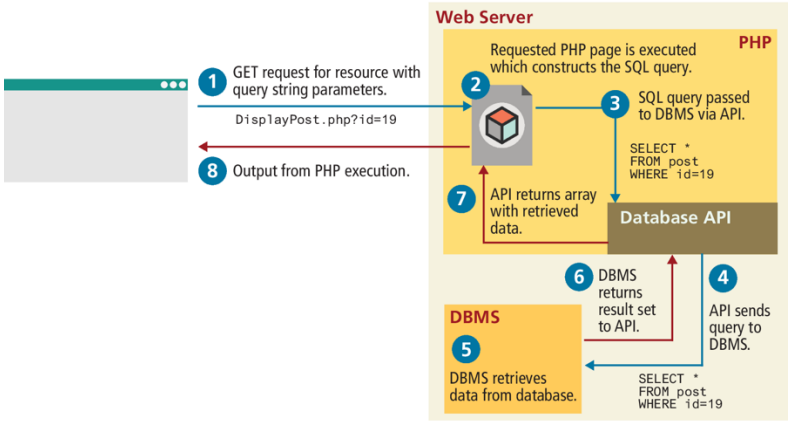
The Role of Databases in Web Development

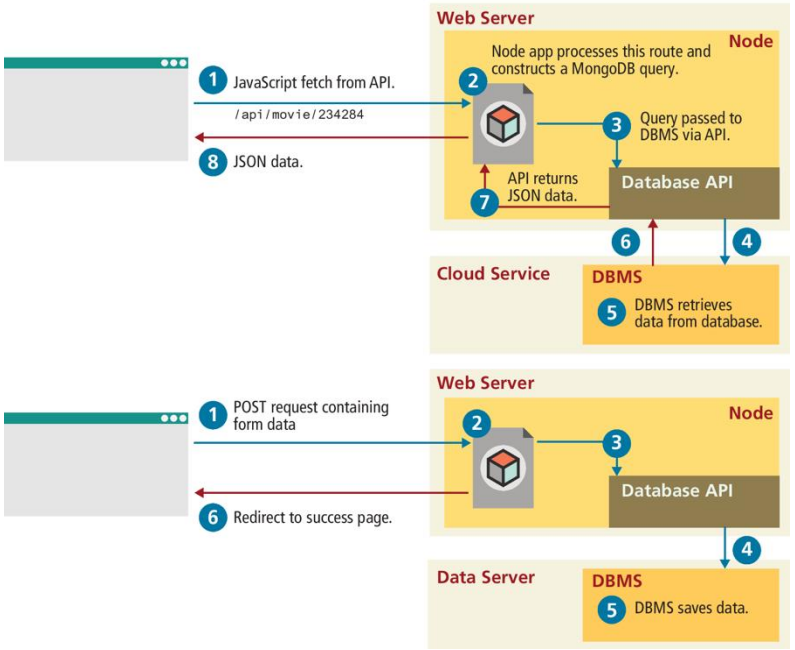
Databases provide a way to implement an important software design principles: namely, that one should *separate that which varies from that which stays the same*.

On the web the visual appearance (i.e., the HTML and CSS) is that which *stays the same*, while the data content is *that which varies*.



How websites use databases





Managing Databases

Running the SQLite lab exercises for PHP and Node, you don't actually have to install anything, since it is a file-based, in-memory database.

To run the PHP exercises in this chapter's lab, you will need access to MySQL. If you have installed XAMPP to run your PHP, MySQL is already installed.

To run the Node exercises in this chapter, you will either need to install MongoDB or make use of a cloud service such as MongoDB Atlas.

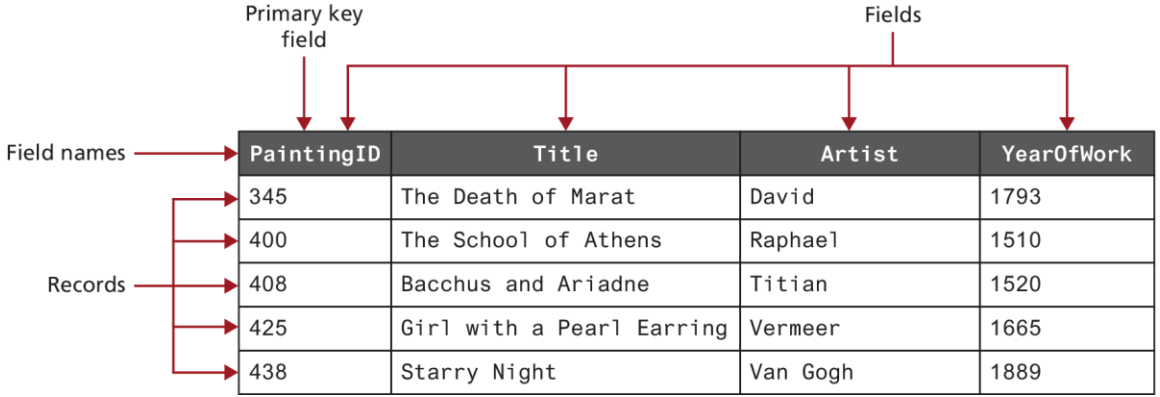
Managing Databases (Tools)

The tools available to you range from the original command-line approach, through to the modern workbench, where an easy-to-use toolset supports the most common operations.

- Command-Line Interface
- phpMyAdmin
- MySQL Workbench
- SQLite Tools
- MongoDB Tools

SQL

A **table** is the principal unit of storage in a database. It is a two-dimensional container for data that consists of **records** (rows); each record has the same number of columns. These columns are called **fields**, which contain the actual data. Each table will have a **primary key**—a field (or sometimes combination of fields) that is used to uniquely identify each record in a table.

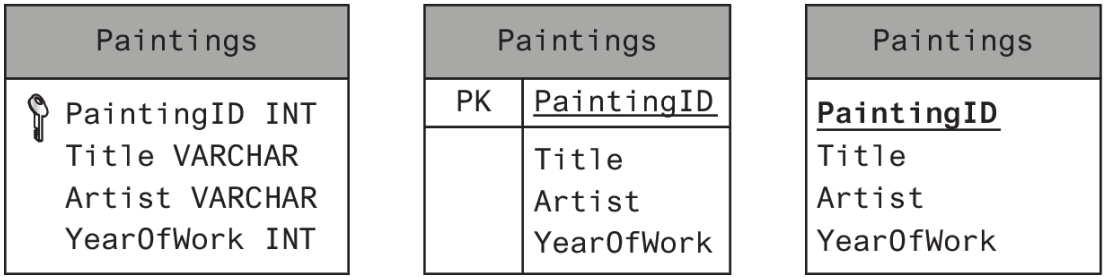


PaintingID	Title	Artist	YearOfWork
345	The Death of Marat	David	1793
400	The School of Athens	Raphael	1510
408	Bacchus and Ariadne	Titian	1520
425	Girl with a Pearl Earring	Vermeer	1665
438	Starry Night	Van Gogh	1889

Table Design

Data types that are akin to those in a statically typed programming language and contribute to data integrity. (BIT,BLOB,CHAR(n), DATE,FLOAT,INT,VARCHAR(n))

As we discuss database tables and their design, it will be helpful to have a condensed way to visually represent a table. It is normally enough to see the field names, and perhaps their data types.



Foreign Key

Foreign key relates a field in one table to a primary key in another table

Tables that are linked via foreign keys are said to have a relationship.

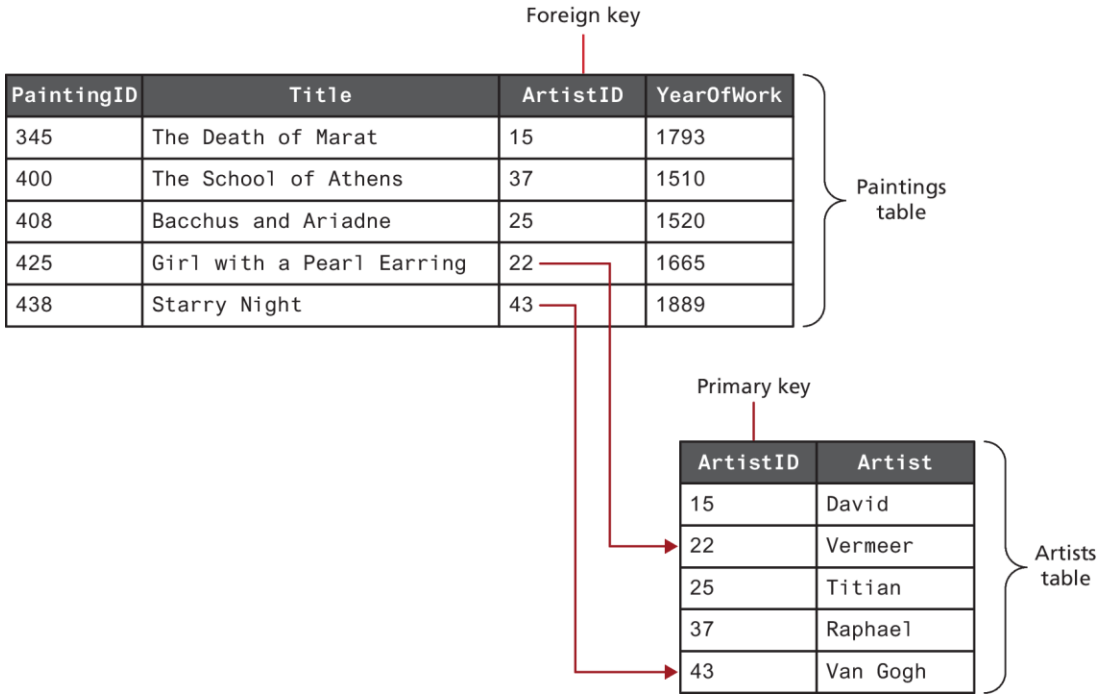
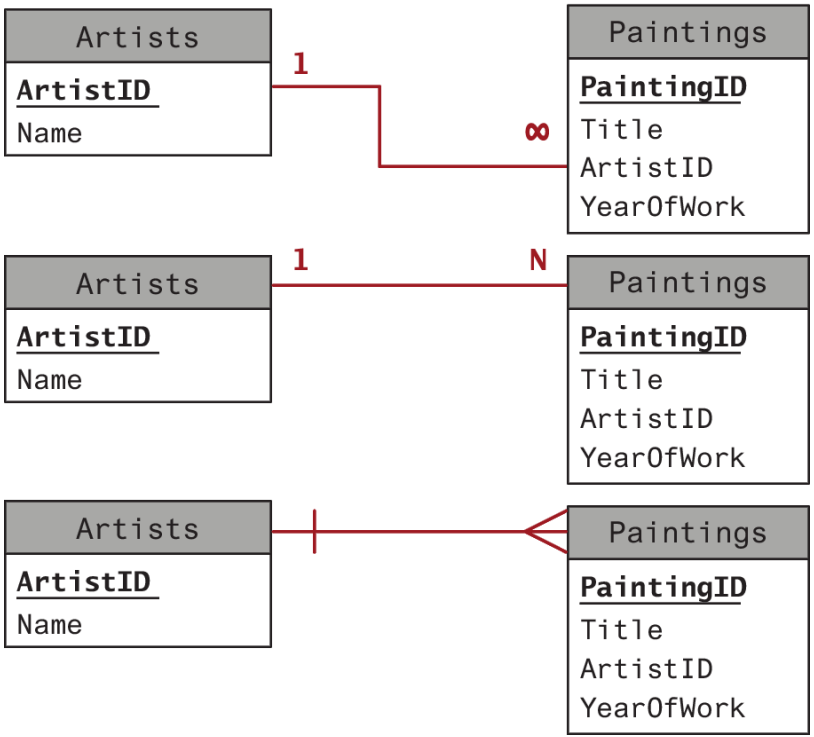


Table relationships

Most often, two related tables will be in a **one-to-many relationship**.

There are two other table relationships: the **one-to-one relationship** and the **many-to-many relationship**.



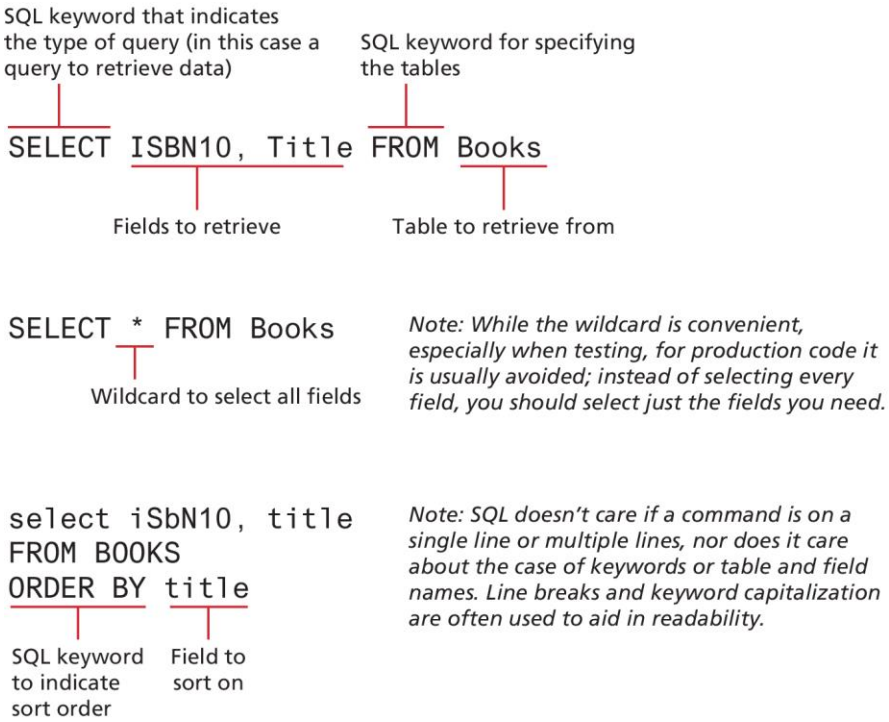
Composite Key

- Note that in this example, the two foreign keys in the intermediate table are combined to create a **composite key**. Alternatively, the intermediate table could contain a separate primary key field.

SELECT Statement

The SELECT statement is used to retrieve data from the database. The term **query** is sometimes used as a synonym for running a SELECT statement

The result of a SELECT statement is a block of data typically called a **result set** which can be ordered



WHERE Clause

The WHERE keyword is used to supply a comparison expression that the data must match in order for a record to be included in the result set.

```
SELECT isbn10, title FROM books
WHERE copyrightYear > 2010
```

SQL keyword that indicates to return only those records whose data matches the following criteria expression

Expressions take form: field *operator* value

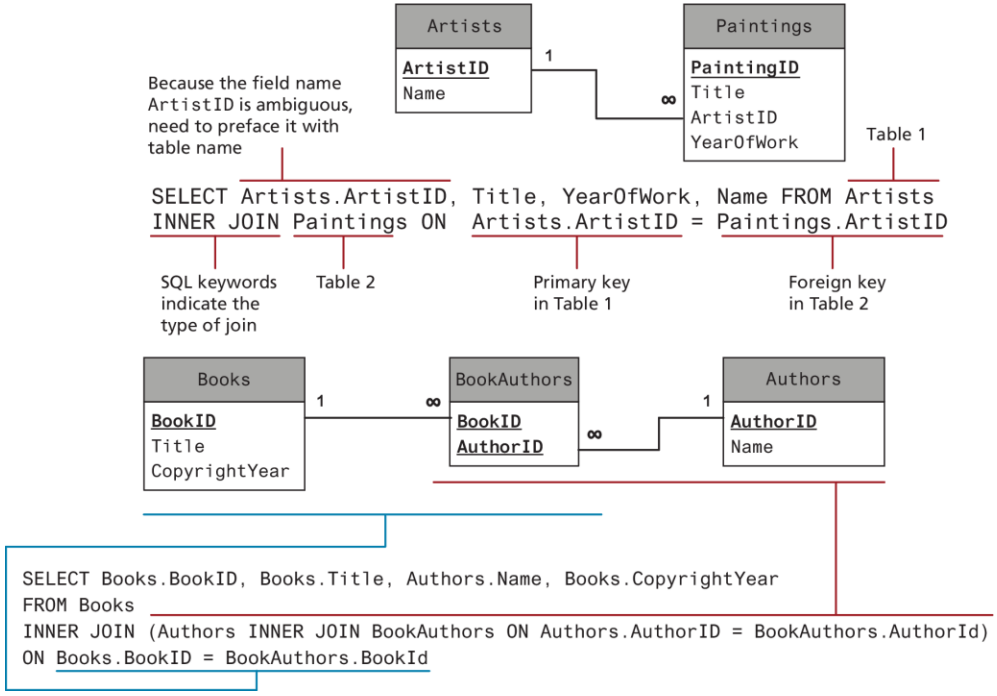
```
SELECT isbn10, title FROM books
WHERE category = 'Math' AND copyrightYear = 2014
```

Comparisons with strings require string literals (single or double quote)

Join

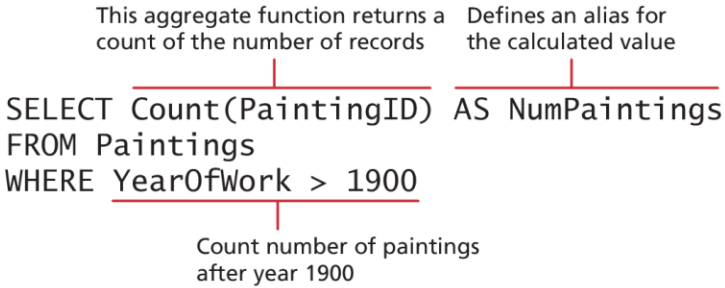
Retrieving data from multiple tables is more complex and requires the use of a **join**.

When two tables are joined via an **inner join**, records are returned if there is matching data (typically from a primary key in one table and a foreign key in the other) in both tables



Grouping

When you don't want every record in your table but instead want to perform some type of calculation on multiple records and then return the results you use one or more **aggregate functions** such as SUM() or COUNT(); these are often used in conjunction with the GROUP BY keywords.



Note: This SQL statement returns a single record with a single value in it.

NumPaintings
745

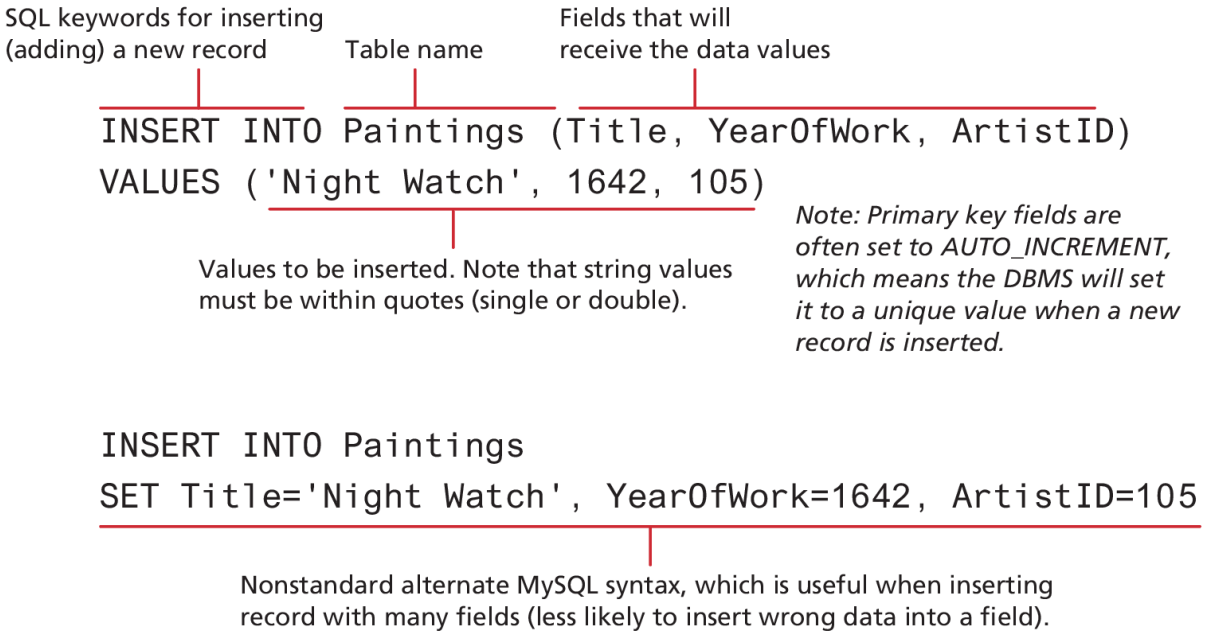
```
SELECT Nationality, Count(ArtistID) AS NumArtists
FROM Artists
GROUP BY Nationality
```

SQL keywords to group output by specified fields

Note: This SQL statement returns as many records as there are unique values in the group-by field.

Nationality	NumArtists
Belgium	4
England	15
France	36
Germany	27
Italy	53

INSERT Statements



UPDATE and DELETE Statements

```
UPDATE Paintings
SET Title='Night Watch', YearOfWork=1642, ArtistID=105
WHERE PaintingID=54
```

It is essential to specify which record to update, otherwise it will update all the records!

Specify the values for each updated field. *Note: Primary key fields that are AUTO_INCREMENT cannot have their values updated.*

```
DELETE FROM Paintings
WHERE PaintingID=54
```

It is essential to specify which record to delete, otherwise it will delete all the records!

Transactions

A **transaction** refers to a sequence of steps that are treated as a single unit, and provide a way to gracefully handle errors and keep your data properly consistent when errors do occur.

Local transaction support in the DBMS can handle the problem of an error with START TRANSACTION, COMMIT, and ROLLBACK commands.

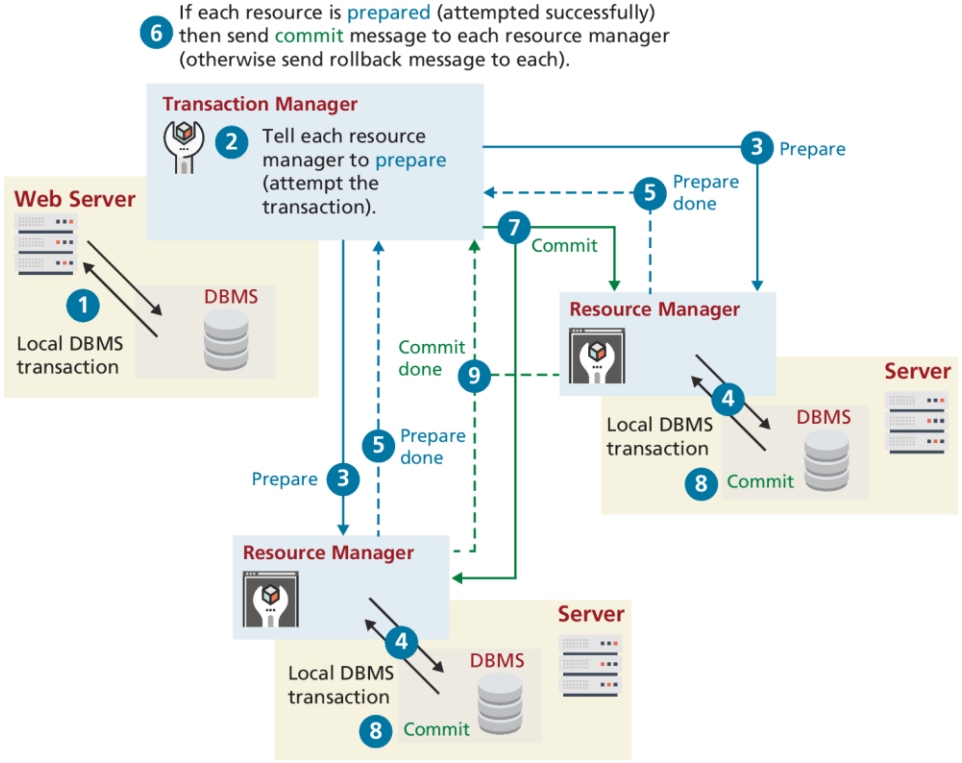
```
/* By starting the transaction, all database modifications within will only be permanently saved in the database if they all work */
START TRANSACTION
INSERT INTO orders . . .
INSERT INTO orderDetails . . .
UPDATE inventory . . .
/* if we have made it here everything has worked so commit changes */
COMMIT
```

LISTING 14.2 SQL commands for transaction processing

Distributed Transactions

Transactions involving multiple hosts, several of which we may have no control over; are typically called **distributed transactions**.

A distributed transaction not only requires local database writes, but also the involvement of an external credit card processor, an external legacy ordering system, and an external shipping system. Because there are multiple external resources involved, distributed transactions are much more complicated than local transactions.



Data Definition Statements

All of the SQL examples that you will use in this book are examples of the data manipulation language features of SQL, that is, SELECT, UPDATE, INSERT, and DELETE. There is also a **Data Definition Language (DDL)** in SQL, which is used for creating tables, modifying the structure of a table, deleting tables, and creating and deleting databases.

Most tools such as the phpMyAdmin, offer interfaces that allow you to manipulate table indirectly through a GUI.

Database Indexes and Efficiency

Consider the worst-case scenario for searching where we compare a query against every single record. If there are *n* elements, we say it takes **O(n)** time to do a search (we would say “Order of *n*”).

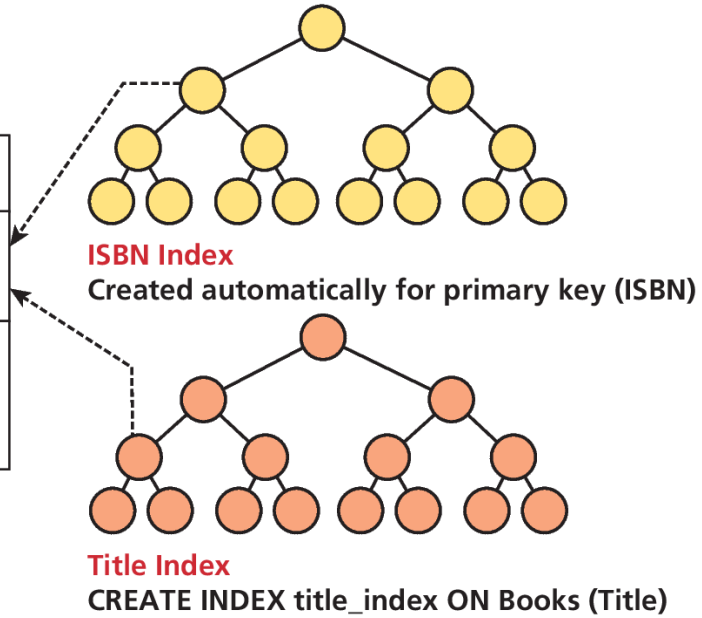
In comparison, a balanced **binary tree** data structure can be searched in **O(log2 n)** time.

It is possible to achieve **O(1)** search speed—that is, one operation to find the result—with a **hash table** data structure.

No matter which data structure is used, the application of that structure to ensure results are quickly accessible is called an **index**.

Database Index

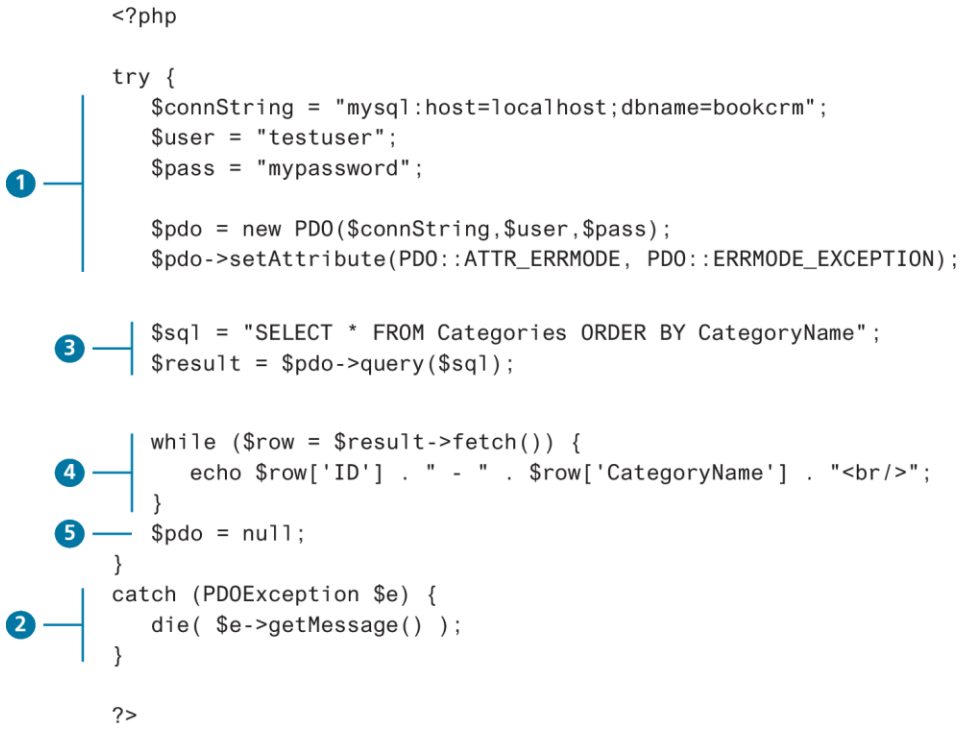
ISBN	Title	Year
0132569035	Computer Science: An Overview, 11/E	2012
0132828936	Fluency with Information Technology: Skills, Concepts, and Capabilities	2013



Working with SQL in PHP

With PDO, the basic database connection algorithm is:

1. Connect to the database.
2. Handle connection errors.
3. Execute the SQL query.
4. Process the results.
5. Free resources and close connection.



Connecting to a Database

With MySQL databases, you supply:

- the host or URL of the database server, the
- database name, and
- the database user name and password.

With SQLite databases, you only need to supply the path to the file:

```
$pdo = new PDO('sqlite:./movies.db');
```

```
// modify these variables for your installation
$connectionString =
"mysql:host=localhost;dbname=bookcrm";
// you may need to add this if db has UTF data
$connectionString .= ";charset=utf8mb4;";
$user = "testuser";
$pass = "mypassword";
$pdo = new PDO($connectionString,
                        $user, $pass);
```

LISTING 14.4 Connecting to a database with PDO (object-oriented)

Storing Connection Details

A common solution is to store connection details in defined constants within a file named **config.inc.php.**

```
require_once('protected/config.inc.php');
$pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
```

```
<?php
define('DBHOST', 'localhost');
define('DBNAME', 'bookcrm');
define('DBUSER', 'testuser');
define('DBPASS', 'mypassword');
define('DBCONNSTRING',"mysql:host=". DBHOST. ";dbname=". DBNAME);
?>
```

LISTING 14.2 Defining connection details via constants in a separate file (config.inc.php)

Handling Connection Errors

Unfortunately not every database connection always works. The approach in PDO for handling connection errors uses try...catch exception- handling blocks.

It should be noted that PDO has three different error-handling approaches/modes.

```
try {
$pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
...
}
catch (PDOException $e) {
die( $e->getMessage() );
}
```

LISTING 14.7 Handling connection errors with PDO

Executing the Query

If the connection to the database is successfully created, then you are ready to construct and execute the query.

```
$sql = "SELECT * FROM Categories ORDER BY CategoryName";
$result = $pdo->query($sql);
```

LISTING 14.9 Executing a SELECT query

```
$sql = "DELETE FROM artists WHERE LastName = 'Connolly'";
// returns number of rows that were deleted
$count = $pdo->exec($sql);
```

LISTING 14.10 Executing a DELETE query

Processing the Query Results

If you are running a SELECT query, then you will want to do something with the retrieved result set, such as displaying it, or performing calculations on it, or searching for something in it.

```
$sql = "SELECT * FROM Paintings ORDER BY Title";
$result = $pdo->query($sql);

// fetch a record from result set into an associative array
while ($row = $result->fetch()) {
        // the keys match the field names from the table
        echo $row['ID']. " - ". $row['Title'];
        echo "<br/>";
}
```

LISTING 14.11 Looping through the result set

Fetching from a result set

```
$sql = "select * from Paintings";
$result = $pdo->query($sql);
```

$result
Result set is a type of cursor to the retrieved data

ID	Title	Artist	Year
345	The Death of Marat	David	1793
400	The School of Athens	Raphael	1510
408	Bacchus and Ariadne	Titian	1520
425	Girl with a Pearl Earring	Vermeer	1665
438	Starry Night	Van Gogh	1889

```
$row = $result->fetch()
```

$row
Associative array

ID	Title	Artist	Year	
345	Death of Marat	David	1793	values

keys

Fetching into an Object

Given the following simple class we can have PHP populate an object of type Book

```
class Book {
    public $ID;
    public $Title;
    public $CopyrightYear;
    public $Description;
}
```

```
$sql = "SELECT * FROM Books";
$result = $pdo->query($sql);
// fetch a record into an object of type Book
while ( $b = $result->fetchObject('Book') ) {
// the property names match the table field names
            echo 'ID: '. $b->ID . '<br/>';
            echo 'Title: '. $b->Title . '<br/>';
            echo 'Year: '. $b->CopyrightYear . '<br/>';
            echo 'Description: '. $b->Description . '<br/>';
            echo '<hr>';
}
```

LISTING 14.13 Populating an object from a result set (PDO)

Freeing Resources and Closing Connection

When you are finished retrieving and displaying your requested data, you should release the memory used by any result sets and then close the connection so that the database system can allocate it to another process.

```
try {
$pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
            ...
            // closes connection and frees the resources used by the PDO object
            $pdo = null;
}
```

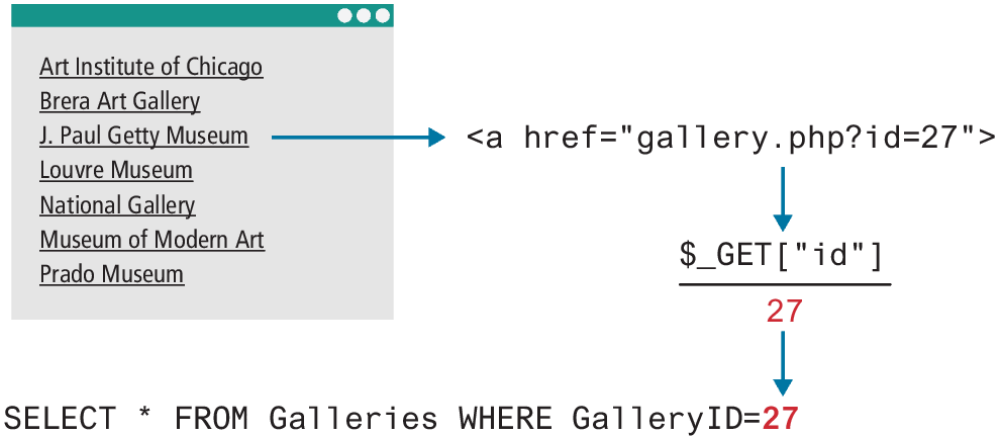
LISTING 14.15 Closing the connection

Working with Parameters

We can use the same page design to display different data records

How does a PHP page "know" which data record to display?

query string parameters




```
$pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
$sql = "SELECT * FROM Galleries WHERE GalleryID=". $_GET["id"];
$result = $pdo->query($sql);
```

LISTING 14.16 Integrating user input into a query (first attempt)

Sanitizing User Data

The last example is vulnerable to SQL injection attack (Chapter 16)

Sanitization uses capabilities built into database systems to remove any special characters from a desired piece of text.

In MySQL, user inputs can be partly sanitized using the **quote()** method.

However, it is recommended that you use **prepared statements**.

Prepared Statements

A **prepared statement** is actually a way to improve performance for queries that need to be executed multiple times.

```
// retrieve parameter value from query string
$id = $_GET['id’];

/* method 1 – notice the ? parameter */
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID = ?";
$statement = $pdo->prepare($sql);
$statement->bindValue(1, $id); // bind to the 1st ? parameter
$statement->execute();

/* method 2 */
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID = :id";
$statement = $pdo->prepare($sql);
$statement->bindValue(':id', $id);
$statement->execute();
```

LISTING 14.17 Using a prepared statement

Named Parameters

A **named parameter** assigns labels in prepared SQL statements which are then explicitly bound to variables in PHP, reducing opportunities for error.

It is also possible to pass in parameter values within an array to the execute() method and cut out the calls to bindValue()

```
/* technique named parameters */
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId,
ProductionStatusId, TrimSize, Description) VALUES (:isbn,
:title,:year,:imprint,:status,:size,:desc) ";
$statement = $pdo->prepare($sql);
$statement->bindValue(':isbn', $_POST['isbn']);
$statement->bindValue(':title', $_POST['title']);
$statement->bindValue(':year', $_POST['year']);
$statement->bindValue(':imprint', $_POST['imprint']);
$statement->bindValue(':status', $_POST['status']);
$statement->bindValue(':size', $_POST['size']);
$statement->bindValue(':desc', $_POST['desc']);
$statement->execute();
```

LISTING 14.18 Using names parameters (part b)

Accessing MySQL in PHP

```
$sql = "UPDATE Categories SET CategoryName='Web' WHERE CategoryName='Business'";

$count = $pdo->exec($sql);

echo "<p>Updated " . $count . " rows</p>";
```

Accessing MySQL in PHP

```
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId,

ProductionStatusId, TrimSize, Description) VALUES (?,?,?,?,?,?,?)";

$statement = $pdo->prepare($sql);

$statement->bindValue(1, $_POST['isbn']);

$statement->bindValue(2, $_POST['title']);

$statement->bindValue(3, $_POST['year']);

$statement->bindValue(4, $_POST['imprint']);

$statement->bindValue(5, $_POST['status']);

$statement->bindValue(6, $_POST['size']);

$statement->bindValue(7, $_POST['desc']);

$statement->execute();
```

Accessing MySQL in PHP

```
/* can pass an array, to be used in order */

$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId,

ProductionStatusId, TrimSize, Description) VALUES (?,?,?,?,?,?,?)";

$statement = $pdo->prepare($sql);

$statement->execute (array($_POST['isbn'], $_POST['title'],$_POST['year'],
$_POST['imprint'], $_POST['status'], $_POST['size'],$_POST['desc']));
```

Accessing MySQL in PHP

```
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId, ProductionStatusId, TrimSize, Description) VALUES (:isbn,
:title, :year, :imprint, :status, :size, :desc) ";

$statement = $pdo->prepare($sql);

$statement->bindValue(':isbn', $_POST['isbn']);

$statement->bindValue(':title', $_POST['title']);

$statement->bindValue(':year', $_POST['year']);

$statement->bindValue(':imprint', $_POST['imprint']);

$statement->bindValue(':status', $_POST['status']);

$statement->bindValue(':size', $_POST['size']);

$statement->bindValue(':desc', $_POST['desc']);

$statement->execute();
```

Accessing MySQL in PHP

```
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId, ProductionStatusId,
TrimSize, Description) VALUES (:isbn, :title, :year, :imprint, :status, :size, :desc) ";

$statement = $pdo->prepare($sql);

$statement->execute(array(':isbn' => $_POST['isbn'],
                          ':title'=> $_POST['title'],
                          ':year'=> $_POST['year'],
                          ':imprint'=>  $_POST['imprint'],
                          ':status'=> $_POST['status'],
                          ':size'=> $_POST['size']
                          ':desc'=> $_POST['desc']));
```

Using Transactions

```
$pdo = new PDO($connString,$user,$pass);
// turn on exceptions so that exception is thrown if error occurs
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
...
try {
                    // begin a transaction
                    $pdo->beginTransaction();
                                $pdo->exec("INSERT INTO Categories (CategoryName) VALUES ('Philosophy')");
                                $pdo->exec("INSERT INTO Categories (CategoryName) VALUES ('Art')");
                                // if we arrive here, it means that no exception was thrown
                                // which means no query has failed, so we can commit the transaction
                                $pdo->commit();
} catch (Exception $e) {
// we must rollback the transaction since an error occurred
// with insert
$pdo->rollback();
}
```

LISTING 14.20 Using transactions (PDO)

Designing Data Access

Database details such as connection strings and table and field names are examples of externalities. These details tend to change over the life of a web application.

Initially, the database for our website might be a SQLite database on our development machine; later it might change to a MySQL database on a data server, and even later, to a relational cloud service. Ideally, with each change in our database infrastructure, we would have to change very little in our code base.

One simple step might be to extract all PDO code into separate functions or classes and use those instead.

Designing Data Access (ii)

```
class DatabaseHelper {
public static function createConnection($values=array()) {
        $connString = $values[0];
        $user = $values[1];
        $password = $values[2];
        $pdo = new PDO($connString,$user,$password);
        $pdo->setAttribute(PDO::ATTR_ERRMODE,
                        PDO::ERRMODE_EXCEPTION);
        $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE,
                        PDO::FETCH_ASSOC);
        return $pdo;
}
public static function runQuery($pdo, $sql, $parameters=array())
{
        // Ensure parameters are in an array
        if (!is_array($parameters)) {
                $parameters = array($parameters);
        }
        $statement = null;
        If (count($parameters) > 0) {
                // Use a prepared statement if parameters
                $statement = $pdo->prepare($sql);
                $executedOk = $statement->execute($parameters);
                if (! $executedOk) {
                throw new PDOException;
        }
        } else {
                // Execute a normal query
                $statement = $pdo->query($sql);
                if (!$statement) {
                        throw new PDOException;
                }
        }
return $statement;
}
} //end class
```

LISTING 14.21 Encapsulating database access via a helper class

Designing Data Access (iii)

```
try {
        $conn = DatabaseHelper::createConnectionInfo(array(DBCONNECTION, DBUSER, DBPASS));
        $sql = "SELECT * FROM Paintings ";
        $paintings = DatabaseHelper::runQuery($conn, $sql, null);
        foreach ($paintings as $p) {
                echo $p["Title"];
        }
        $sql = "SELECT * FROM Artists WHERE Nationality=?";
        $artists = DatabaseHelper::runQuery($conn, $sql, Array("France"));
}
```

Illustrates two example uses of this class. While an improvement, we still have a database dependency in this code with the SQL statements and field names.

A table gateway

```
class PaintingDB {
private static $baseSQL = "SELECT * FROM Paintings ";
public function __construct($connection) {
        $this->pdo = $connection;
}
public function getAll() {
        $sql = self::$baseSQL;
        $statement = DatabaseHelper::runQuery($this->pdo,
                                                $sql, null);
        return $statement->fetchAll();
}
public function findById($id) {
        $sql = self::$baseSQL . " WHERE PaintingID=?";
        $statement = DatabaseHelper::runQuery($this->pdo, $sql,
                                                Array($id));
        return $statement->fetch();
}
public function getAllForArtist($artistID) {
        $sql = self::$baseSQL . " WHERE Paintings.ArtistID=?";
        $statement = DatabaseHelper::runQuery($this->pdo, $sql,
                        Array($artistID));
        return $statement->fetchAll();
}
public function getAllForGallery($galleryID) {
        $sql = self::$baseSQL . " WHERE Paintings.GalleryID=?";
        $statement = DatabaseHelper::runQuery($this->pdo, $sql,
                                                Array($galleryID));
        return $statement->fetchAll();
}
}
```

LISTING 14.22 Sample gateway class for painting table

NoSQL Databases

NoSQL (which stands for Not-only-SQL) is category of database software that describes a style of database that doesn't use the relational table model of normal SQL databases.

NoSQL databases rely on a different set of ideas for data modeling that put fast retrieval ahead of other considerations like consistency.

Systems like DynamoDB, Firebase, and MongoDB now power thousands of sites including household names like Netflix, eBay, Instagram, Forbes, Facebook, and others.

Why (and Why Not) Choose NoSQL?

NoSQL systems handle huge datasets better than relational systems.

NoSQL databases aren't the best answer for all scenarios. SQL databases use schemas for a very good reason: they ensure data consistency and data integrity.

The data in most NoSQL database systems is identified by a unique key. The key-value organization often results in faster retrieval of data in comparison to a relational database

Key-Value Stores

Key-value stores alone are quite straightforward in that every value, whether an integer, string, or other data structure, has an associated key (i.e., they are analogous to PHP associative arrays)

Here every value has a key. This allows fast retrieval through means such as a hash function, and precludes the need for indexes on multiple fields as is the case with SQL

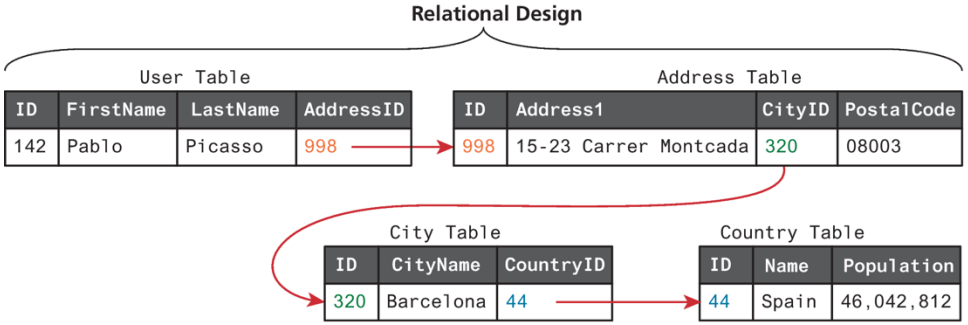
Key	Value
Customer.Name	"Randy"
Price	200.00
ShippingAddress	"4825 Mount Royal Gate SW"
Countries	"Canada","France","Germany","United States"

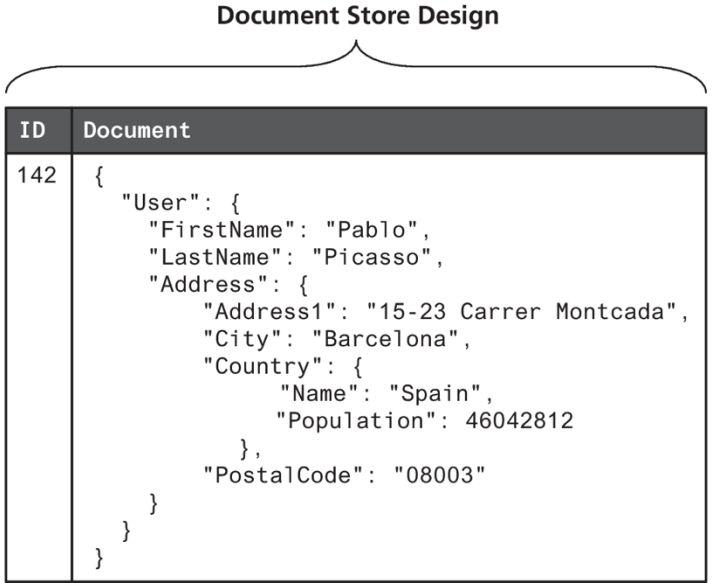
Document Store

Document Stores (also called document-oriented databases) associate keys with values, but unlike key-value stores, they call that value a document.

- A document can be a binary file like a .doc or .pdf or a semi-structured XML or JSON document.
- Most NoSQL systems are of this type. MongoDB, AWS DynamoDB, Google FireBase, and Cloud Datastore are popular examples.

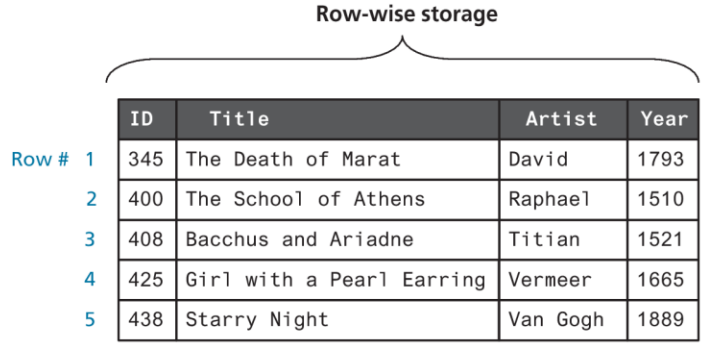
Relational data versus document store data



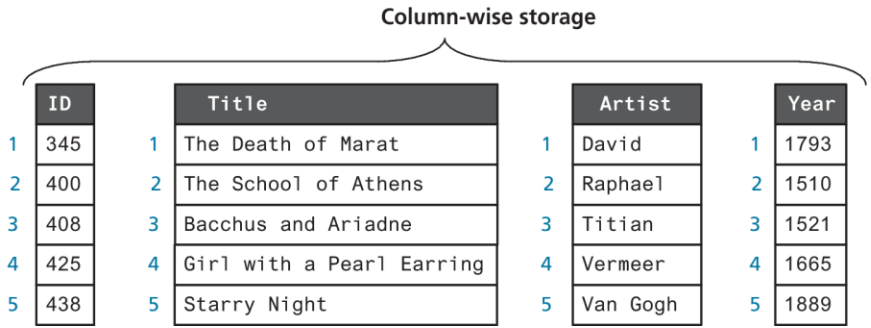


Column Stores

In traditional relational database systems, the data in tables is stored in a row-wise manner. This means that the fundamental unit of data retrieved is a row.



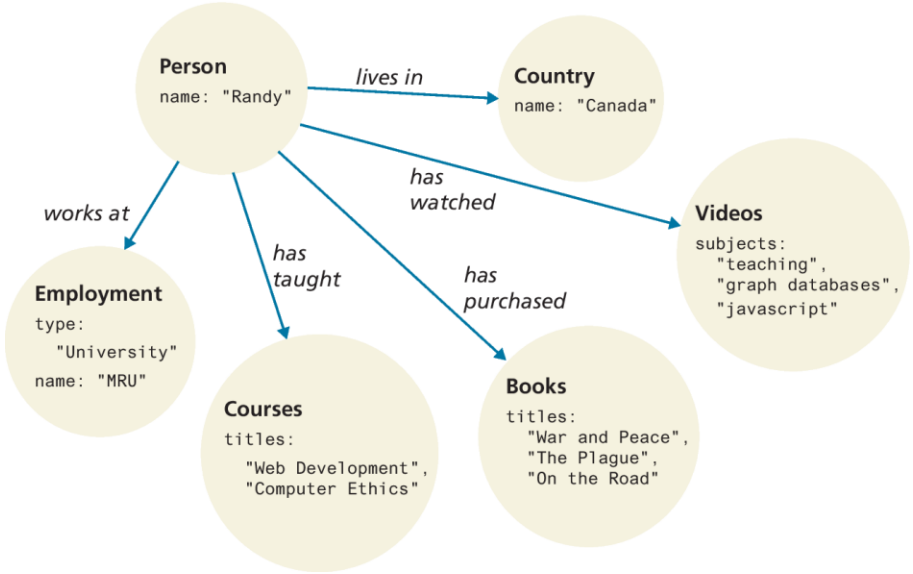
Column Store systems store data by column instead of by row, meaning that fetches retrieve a column of data and retrieving an entire row requires multiple operations.



Graph Stores

In a **Graph Store** system (often simply called graph databases), data is represented as a network or graph of entities and their relationships.

Some examples of graph databases include Neo4j, OrientDB, and RedisGraph.



Working with MongoDB in Node

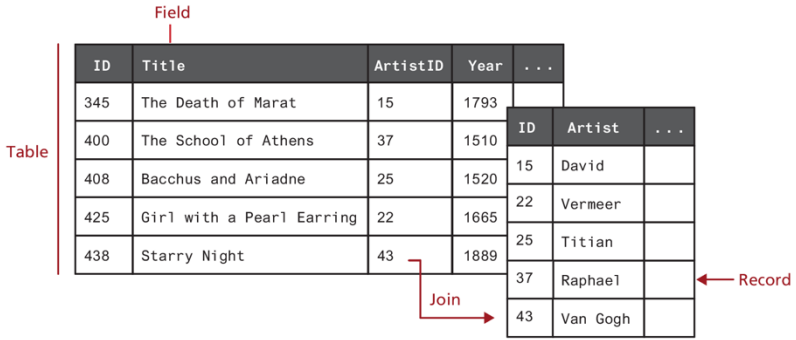
MongoDB MongoDB is an open-source, NoSQL, document-oriented database. It can be used with PHP, it is much more commonly used with Node

You simply package your data as a JSON object, give it to MongoDB, and it stores this object or document as a binary JavaScript object (BSON).

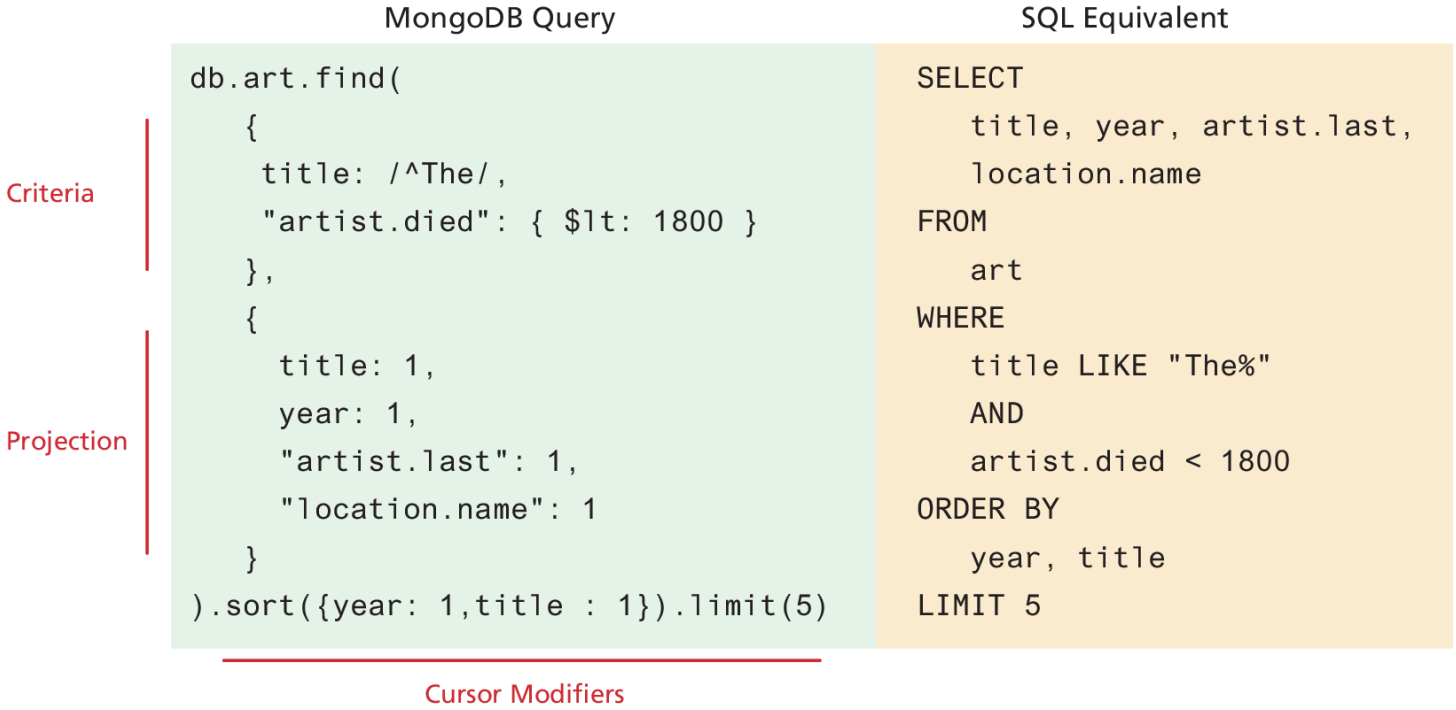
MongoDB does not support transactions

The ability to run on multiple servers means MongoDB can handle large datasets

Comparing relational databases to the MongoDB data model



Comparing a MongoDB query to an SQL query



Working with the MongoDB Shell

```
~/workspace $ mongod
mongod --help for help and startup options
2016-08-03T20:14:00.020+0000 [initandlisten] MongoDB starting : ...
2016-08-03T20:14:00.020+0000 [initandlisten] db version v2.6.11
2016-08-03T20:14:00.020+0000 [initandlisten] git version: ...
...
2016-08-04T17:00:49.737+0000 [initandlisten] waiting for connections on port 27017
```

1 MongoDB daemon process needs to be started in a separate terminal window

```
~/workspace $ mongo
MongoDB shell version: 2.6.11
connecting to: test
> use funwebdev
switched to db funwebdev
>
>
> db.art.insert({"id":438, "title" : "Starry Night"})
WriteResult({ "nInserted" : 1 })
> db.art.insert({id:400, title : "The School of Athens"})
WriteResult({ "nInserted" : 1 })
>
```

2 The MongoDB shell in another window lets you work with the data

Specifies the database to use (if it doesn't exist it gets created)

Specifies the collection to use (if it doesn't exist it gets created)

Adds new document

Quotes around property names are optional

The MongoDB shell is like the JavaScript console: you can write any valid JavaScript code

```
> for (var i=1; i<=10; i++) db.users.insert({Name : "User" + i, Id: i})
>
> db.art.find()
{ "_id" : ObjectId("57a3780476..."), "id" : 438, "title" : "Starry Night" }
{ "_id" : ObjectId("57a378..."), "id" : 400, "title" : "The School of Athens" }
>
> db.art.find().sort({title: 1})
...
> db.art.find({id:400})
...
> db.art.find({ id: {$gte: 400} })
...
> db.art.find( {title: /Night/} )
...
> quit()
~/workspace $
~/workspace $ mongoimport --db funwebdev --collection books --file books.json --jsonArray
connected to: 127.0.0.1
2016-08-04T19:12:28.053+0000 check 9 215
2016-08-04T19:12:28.053+0000 imported 215 objects
~/workspace $
```

returns all data in specified collection

Sorts on title field (1=ascending)

Searches for object with id = 400

Searches for objects with id >= 400

Regular expression search

3 Imports JSON data file into funwebdev database in the collection books

Accessing MongoDB Data in Node.js

```
require('dotenv').config();
console.log(process.env.MONGO_URL);
const mongoose = require('mongoose');
mongoose.connect(process.env.MONGO_URL,
{useNewUrlParser: true, useUnifiedTopology: true});
const db = mongoose.connection;
db.on('error', console.error.bind(console,
                                   'connection error:'));
db.once('open', () => {
 console.log('connected to mongo');
});
```

LISTING 14.23 Connecting to MongoDB using Mongoose

```
const mongoose = require('mongoose');
// define a schema that maps to the structure in MongoDB
const bookSchema = new mongoose.Schema({
        id: Number,
        isbn10: String,
        isbn13: String,
        title: String,
        ...
        },
        category: {
                main: String,
                secondary: String
        }
});
// now create model using this schema that maps to books
collection in database
module.exports = mongoose.model('Book',
bookSchema,'books');
```

LISTING 14.24 Creating a Mongoose model

Web service using MongoDB

```
// get our data model
const Book = require('./models/Book.js');

app.get('/api/books', (req,resp) => {
// use mongoose to retrieve all books
    Book.find({}, function(err, data) {
        if (err) {
            resp.json({ message:
                        'Unable to connect to books' });
        } else {
            // return JSON retrieved by Mongo as response
            resp.json(data);
        }
    });
});
```

```
app.get('/api/books/:isbn', (req,resp) => {
    // use mongoose to retrieve all books from Mongo
    Book.find({isbn10: req.params.isbn},
        function(err, data) {
        if (err) {
            resp.json({ message: 'Book not found' });
        } else {
            resp.json(data);
        }
    });
});
```

LISTING 14.25 Web service using MongoDB data and Mongoose ORM

Key Terms

aggregate functions

binary tree

clickstream

column store

commodity servers

composite key

connection

connection string

database

data integrity

Data Definition Language

(DDL)

data duplication

database normalization

distributed transactions

document stores

failover clustering

fields

foreign key

GraphQL

graph store

hash table

index

inner join

join

key-value stores

local transactions

many-to-many relationship

multiple-master replication

MySQL

named parameter

NoSQL

one-to-many relationship

one-to-one relationship

ORM (Object-Relational Mapping)

phpMyAdmin

prepared statement

primary key

query

record

result set

sanitization

sharding

single-master replication

SQL

SQL script

table

table gateway

transaction

two-phase commit

Copyright