

Extracted from:

Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

This PDF file contains pages extracted from *Seven Databases in Seven Weeks*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

Eric Redmond
and Jim R. Wilson

Edited by Jacquelyn Carter



Seven Databases in Seven Weeks

A Guide to Modern Databases
and the NoSQL Movement

Eric Redmond
Jim R. Wilson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Introduction

This is a pivotal time in the database world. For years the relational model has been the *de facto* option for problems big and small. We don't expect relational databases will fade away anytime soon, but people are emerging from the RDBMS fog to discover alternative options, such as schemaless or alternative data structures, simple replication, high availability, horizontal scaling, and new query methods. These options are collectively known as *NoSQL* and make up the bulk of this book.

In this book, we explore seven databases across the spectrum of database styles. In the process of reading the book, you will learn the various functionality and trade-offs each database has—durability vs. speed, absolute vs. eventual consistency, and so on—and how to make the best decisions for your use cases.

1.1 It Starts with a Question

The central question of *Seven Databases in Seven Weeks* is this: what database or combination of databases best resolves your problem? If you walk away understanding how to make that choice, given your particular needs and resources at hand, we're happy.

But to answer that question, you'll need to understand your options. For that, we'll take you on a deep dive into each of seven databases, uncovering the good parts and pointing out the not so good. You'll get your hands dirty with CRUD, flex your schema muscles, and find answers to these questions:

- *What type of datastore is this?* Databases come in a variety of genres, such as relational, key-value, columnar, document-oriented, and graph. Popular databases—including those covered in this book—can generally be grouped into one of these broad categories. You'll learn about each

type and the kinds of problems for which they're best suited. We've specifically chosen databases to span these categories including one relational database (Postgres), two key-value stores (Riak, Redis), a column-oriented database (HBase), two document-oriented databases (MongoDB, CouchDB), and a graph database (Neo4J).

- *What was the driving force?* Databases are not created in a vacuum. They are designed to solve problems presented by real use cases. RDBMS databases arose in a world where query flexibility was more important than flexible schemas. On the other hand, column-oriented datastores were built to be well suited for storing large amounts of data across several machines, while data relationships took a backseat. We'll cover cases in which to use each database and related examples.
- *How do you talk to it?* Databases often support a variety of connection options. Whenever a database has an interactive command-line interface, we'll start with that before moving on to other means. Where programming is needed, we've stuck mostly to Ruby and JavaScript, though a few other languages sneak in from time to time—like PL/pgSQL (Postgres) and Gremlin (Neo4J). At a lower level, we'll discuss protocols like REST (CouchDB, Riak) and Thrift (HBase). In the final chapter, we present a more complex database setup tied together by a Node.js JavaScript implementation.
- *What makes it unique?* Any datastore will support writing data and reading it back out again. What else it does varies greatly from one to the next. Some allow querying on arbitrary fields. Some provide indexing for rapid lookup. Some support ad hoc queries; for others, queries must be planned. Is schema a rigid framework enforced by the database or merely a set of guidelines to be renegotiated at will? Understanding capabilities and constraints will help you pick the right database for the job.
- *How does it perform?* How does this database function and at what cost? Does it support sharding? How about replication? Does it distribute data evenly using consistent hashing, or does it keep like data together? Is this database tuned for reading, writing, or some other operation? How much control do you have over its tuning, if any?
- *How does it scale?* Scalability is related to performance. Talking about scalability without the context of what you want to *scale to* is generally fruitless. This book will give you the background you need to ask the right questions to establish that context. While the discussion on *how* to scale each database will be intentionally light, in these pages you'll find out

whether each datastore is geared more for horizontal scaling (MongoDB, HBase, Riak), traditional vertical scaling (Postgres, Neo4J, Redis), or something in between.

Our goal is not to guide a novice to mastery of any of these databases. A full treatment of any one of them could (and does) fill entire books. But by the end you should have a firm grasp of the strengths of each, as well as how they differ.

1.2 The Genres

Like music, databases can be broadly classified into one or more styles. An individual song may share all of the same notes with other songs, but some are more appropriate for certain uses. Not many people blast Bach's *Mass in B Minor* out an open convertible speeding down the 405. Similarly, some databases are better for some situations over others. The question you must always ask yourself is not "Can I use this database to store and refine this data?" but rather, "Should I?"

In this section, we're going to explore five main database genres. We'll also take a look at the databases we're going to focus on for each genre.

It's important to remember that most of the data problems you'll face could be solved by most or all of the databases in this book, not to mention other databases. The question is less about whether a given database style could be shoehorned to model your data and more about whether it's the best fit for your problem space, your usage patterns, and your available resources. You'll learn the art of divining whether a database is intrinsically useful to you.

Relational

The relational model is generally what comes to mind for most people with database experience. Relational database management systems (RDBMSs) are set-theory-based systems implemented as two-dimensional tables with rows and columns. The canonical means of interacting with an RDBMS is by writing queries in Structured Query Language (SQL). Data values are typed and may be numeric, strings, dates, uninterpreted blobs, or other types. The types are enforced by the system. Importantly, tables can join and morph into new, more complex tables, because of their mathematical basis in relational (set) theory.

There are lots of open source relational databases to choose from, including MySQL, H2, HSQLDB, SQLite, and many others. The one we cover is in [Chapter 2, PostgreSQL, on page ?](#).

PostgreSQL

Battle-hardened PostgreSQL is by far the oldest and most robust database we cover. With its adherence to the SQL standard, it will feel familiar to anyone who has worked with relational databases before, and it provides a solid point of comparison to the other databases we'll work with. We'll also explore some of SQL's unsung features and Postgres's specific advantages. There's something for everyone here, from SQL novice to expert.

Key-Value

The key-value (KV) store is the simplest model we cover. As the name implies, a KV store pairs keys to values in much the same way that a map (or hashtable) would in any popular programming language. Some KV implementations permit complex value types such as hashes or lists, but this is not required. Some KV implementations provide a means of iterating through the keys, but this again is an added bonus. A filesystem could be considered a key-value store, if you think of the file path as the key and the file contents as the value. Because the KV moniker demands so little, databases of this type can be incredibly performant in a number of scenarios but generally won't be helpful when you have complex query and aggregation needs.

As with relational databases, many open source options are available. Some of the more popular offerings include memcached (and its cousins memcachedb and membase), Voldemort, and the two we cover in this book: Redis and Riak.

Riak

More than a key-value store, *Riak*—covered in [Chapter 3, Riak, on page ?](#)—embraces web constructs like HTTP and REST from the ground up. It's a faithful implementation of Amazon's Dynamo, with advanced features such as vector clocks for conflict resolution. Values in Riak can be anything, from plain text to XML to image data, and relationships between keys are handled by named structures called *links*. One of the lesser known databases in this book, Riak, is rising in popularity, and it's the first one we'll talk about that supports advanced querying via mapreduce.

Redis

Redis provides for complex datatypes like sorted sets and hashes, as well as basic message patterns like publish-subscribe and blocking queues. It also has one of the most robust query mechanisms for a KV store. And by caching writes in memory before committing to disk, Redis gains amazing performance in exchange for increased risk of data loss in the case of a hardware failure. This characteristic makes it a good fit for caching noncritical data and for acting as a message broker. We leave it until the end—see [Chapter 8, Redis, on page ?](#)—so we can build a multidatabase application with Redis and others working together in harmony.

Columnar

Columnar, or column-oriented, databases are so named because the important aspect of their design is that data from a given column (in the two-dimensional table sense) is stored together. By contrast, a row-oriented database (like an RDBMS) keeps information about a row together. The difference may seem inconsequential, but the impact of this design decision runs deep. In column-oriented databases, adding columns is quite inexpensive and is done on a row-by-row basis. Each row can have a different set of columns, or none at all, allowing tables to remain *sparse* without incurring a storage cost for null values. With respect to structure, columnar is about midway between relational and key-value.

In the columnar database market, there's somewhat less competition than in relational databases or key-value stores. The three most popular are HBase (which we cover in [Chapter 4, HBase, on page ?](#)), Cassandra, and Hypertable.

HBase

This column-oriented database shares the most similarities with the relational model of all the nonrelational databases we cover. Using Google's BigTable paper as a blueprint, HBase is built on Hadoop (a mapreduce engine) and designed for scaling horizontally on clusters of commodity hardware. HBase makes strong consistency guarantees and features tables with rows and columns—which should make SQL fans feel right at home. Out-of-the-box support for versioning and compression sets this database apart in the “Big Data” space.

Document

Document-oriented databases store, well, documents. In short, a document is like a hash, with a unique ID field and values that may be any of a variety of types, including more hashes. Documents can contain nested structures,

and so they exhibit a high degree of flexibility, allowing for variable domains. The system imposes few restrictions on incoming data, as long as it meets the basic requirement of being expressible as a document. Different document databases take different approaches with respect to indexing, ad hoc querying, replication, consistency, and other design decisions. Choosing wisely between them requires understanding these differences and how they impact your particular use cases.

The two major open source players in the document database market are MongoDB, which we cover in [Chapter 5, *MongoDB*, on page ?](#), and CouchDB, covered in [Chapter 6, *CouchDB*, on page ?](#).

MongoDB

MongoDB is designed to be *huge* (the name *mongo* is extracted from the word *humongous*). Mongo server configurations attempt to remain consistent—if you write something, subsequent reads will receive the same value (until the next update). This feature makes it attractive to those coming from an RDBMS background. It also offers atomic read-write operations such as incrementing a value and deep querying of nested document structures. Using JavaScript for its query language, MongoDB supports both simple queries and complex mapreduce jobs.

CouchDB

CouchDB targets a wide variety of deployment scenarios, from the datacenter to the desktop, on down to the smartphone. Written in Erlang, CouchDB has a distinct ruggedness largely lacking in other databases. With nearly incorruptible data files, CouchDB remains highly available even in the face of intermittent connectivity loss or hardware failure. Like Mongo, CouchDB's native query language is JavaScript. Views consist of mapreduce functions, which are stored as documents and replicated between nodes like any other data.

Graph

One of the less commonly used database styles, graph databases excel at dealing with highly interconnected data. A graph database consists of nodes and relationships between nodes. Both nodes and relationships can have properties—key-value pairs—that store data. The real strength of graph databases is traversing through the nodes by following relationships.

In [Chapter 7, *Neo4J*, on page ?](#), we discuss the most popular graph database today, Neo4J.

Neo4J

One operation where other databases often fall flat is crawling through self-referential or otherwise intricately linked data. This is exactly where Neo4J shines. The benefit of using a graph database is the ability to quickly traverse nodes and relationships to find relevant data. Often found in social networking applications, graph databases are gaining traction for their flexibility, with Neo4j as a pinnacle implementation.

Polyglot

In the wild, databases are often used alongside other databases. It's still common to find a lone relational database, but over time it is becoming popular to use several databases together, leveraging their strengths to create an ecosystem that is more powerful, capable, and robust than the sum of its parts. This practice is known as *polyglot persistence* and is a topic we consider further in [Chapter 9, Wrapping Up, on page ?](#).

1.3 Onward and Upward

We're in the midst of a Cambrian explosion of data storage options; it's hard to predict exactly what will evolve next. We can be fairly certain, though, that the pure domination of any particular strategy (relational or otherwise) is unlikely. Instead, we'll see increasingly specialized databases, each suited to a particular (but certainly overlapping) set of ideal problem spaces. And just as there are jobs today that call for expertise specifically in administrating relational databases (DBAs), we are going to see the rise of their nonrelational counterparts.

Databases, like programming languages and libraries, are another set of tools that every developer should know. Every good carpenter must understand what's in their toolbox. And like any good builder, you can never hope to be a master without a familiarity of the many options at your disposal.

Consider this a crash course in the workshop. In this book, you'll swing some hammers, spin some power drills, play with some nail guns, and in the end be able to build so much more than a birdhouse. So, without further ado, let's wield our first database: PostgreSQL.