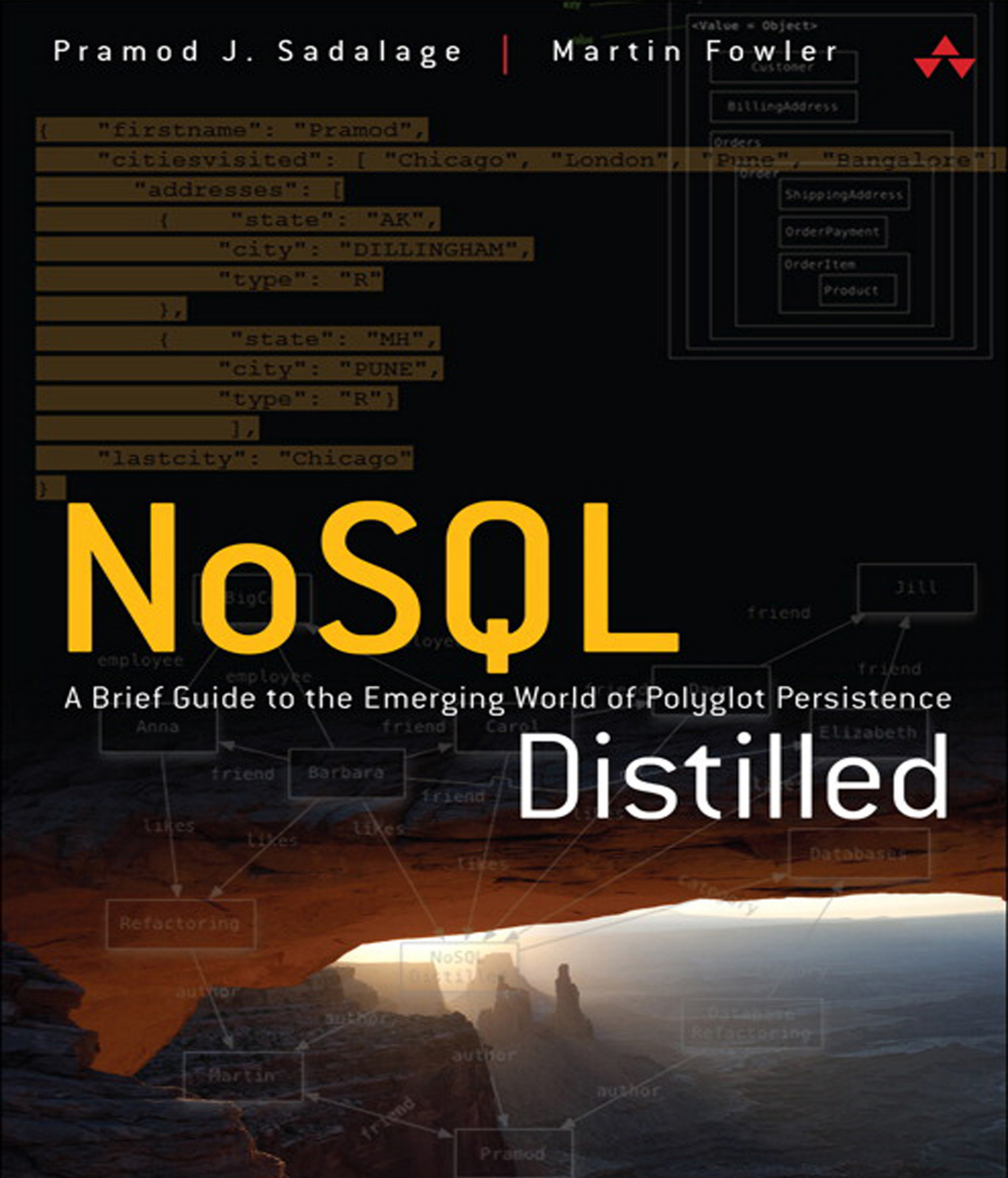
Pramod J. Sadalage | Martin Fowler



NoSQL Distilled

A Brief Guide to the Emerging World of Polyglot Persistence

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

NoSQL Distilled

This page intentionally left blank

NoSQL Distilled

A Brief Guide to the Emerging World of Polyglot Persistence

Pramod J. Sadalage
Martin Fowler

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382–3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Sadalage, Pramod J.
NoSQL distilled : a brief guide to the emerging world of polyglot persistence / Pramod J Sadalage, Martin Fowler.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-82662-6 (pbk. : alk. paper) -- ISBN 0-321-82662-0 (pbk. : alk. paper) 1. Databases--Technological innovations. 2. Information storage and retrieval systems. I. Fowler, Martin, 1963- II. Title.
QA76.9.D32S228 2013
005.74--dc23

ISBN-13: 978-0-321-82662-6
ISBN-10: 0-321-82662-0

ScoutAutomatedPrintCode

For my teachers Gajanan Chinchwadkar, Dattatraya Mhaskar, and Arvind Parchure. You inspired me the most, thank you.

—Pramod

For Cindy

—Martin

This page intentionally left blank

Contents

This page intentionally left blank

Preface

We've spent some twenty years in the world of enterprise computing. We've seen many things change in languages, architectures, platforms, and processes. But through all this time one thing has stayed constant—relational databases store the data. There have been challengers, some of which have had success in some niches, but on the whole the data storage question for architects has been the question of which relational database to use.

There is a lot of value in the stability of this reign. An organization's data lasts much longer than its programs (at least that's what people tell us—we've seen plenty of very old programs out there). It's valuable to have a stable data storage that's well understood and accessible from many application programming platforms.

Now, however, there's a new challenger on the block under the confrontational tag of NoSQL. It's born out of a need to handle larger data volumes which forced a fundamental shift to building large hardware platforms through clusters of commodity servers. This need has also raised long-running concerns about the difficulties of making application code play well with the relational data model.

The term "NoSQL" is very ill-defined. It's generally applied to a number of recent nonrelational databases such as Cassandra, Mongo, Neo4J, and Riak. They embrace schemaless data, run on clusters, and have the ability to trade off traditional consistency for other useful properties. Advocates of NoSQL databases claim that they can build systems that are more performant, scale much better, and are easier to program with.

Is this the first rattle of the death knell for relational databases, or yet another pretender to the throne? Our answer to that is "neither." Relational databases are a powerful tool that we expect to be using for many more decades, but we do see a profound change in that relational databases won't be the only databases in use. Our view is that we are entering a world of Polyglot Persistence where enterprises, and even individual applications, use multiple technologies for data management. As a result, architects will need to be familiar with these technologies and be able to evaluate which ones to use for differing needs.

Had we not thought that, we wouldn't have spent the time and effort writing this book.

This book seeks to give you enough information to answer the question of whether NoSQL databases are worth serious consideration for your future projects. Every project is different, and there's no way we can write a simple decision tree to choose the right data store. Instead, what we are attempting here is to provide you with enough background on how NoSQL databases work, so that you can make those judgments yourself without having to trawl the whole web. We've deliberately made this a small book, so you can get this overview pretty quickly. It won't answer your questions definitively, but it should narrow down the range of options you have to consider and help you understand what questions you need to ask.

Why Are NoSQL Databases Interesting?

We see two primary reasons why people consider using a NoSQL database.

- **Application development productivity.** A lot of application development effort is spent on mapping data between in-memory data structures and a relational database. A NoSQL database may provide a data model that better fits the application's needs, thus simplifying that interaction and resulting in less code to write, debug, and evolve.
- **Large-scale data.** Organizations are finding it valuable to capture more data and process it more quickly. They are finding it expensive, if even possible, to do so with relational databases. The primary reason is that a relational database is designed to run on a single machine, but it is usually more economic to run large data and computing loads on clusters of many smaller and cheaper machines. Many NoSQL databases are designed explicitly to run on clusters, so they make a better fit for big data scenarios.

What's in the Book

We've broken this book up into two parts. The first part concentrates on core concepts that we think you need to know in order to judge whether NoSQL databases are relevant for you and how they differ. In the second part we concentrate more on implementing systems with NoSQL databases.

Chapter 1 begins by explaining why NoSQL has had such a rapid rise—the need to process larger data volumes led to a shift, in large systems, from scaling vertically to scaling horizontally on clusters. This explains an important feature of the data model of many NoSQL databases—the explicit storage of a rich structure of closely related data that is accessed as a unit. In this book we call this kind of structure an *aggregate*.

Chapter 2 describes how aggregates manifest themselves in three of the main data models in NoSQL land: key-value ("Key-Value and Document Data Models," p. 20), document ("Key-Value and Document Data Models," p. 20), and column family ("Column-Family Stores," p. 21) databases. Aggregates provide a natural unit of interaction for many kinds of applications, which both improves running on a cluster and makes it easier to program the data access. Chapter 3 shifts to the downside of aggregates—the difficulty of handling relationships ("Relationships," p. 25) between entities in different aggregates. This leads us naturally to graph databases ("Graph Databases," p. 26), a NoSQL data model that doesn't fit into the aggregate-oriented camp. We also look at the common characteristic of NoSQL databases that operate without a schema ("Schemaless Databases," p. 28)—a feature that provides some greater flexibility, but not as much as you might first think.

Having covered the data-modeling aspect of NoSQL, we move on to distribution: Chapter 4 describes how databases distribute data to run on clusters. This breaks down into sharding ("Sharding," p. 38) and replication, the latter being either leader-follower ("Leader-Follower Replication," p. 40) or peer-to-peer ("Peer-to-Peer Replication," p. 42) replication. With the distribution models defined, we can then move on to the issue of consistency. NoSQL databases provide a more varied range of consistency options than relational databases—which is a consequence of being friendly to clusters. So Chapter 5 talks about how consistency changes for updates ("Update Consistency," p. 47) and reads ("Read Consistency," p. 49), the role of quorums ("Quorums," p. 57), and how even some durability ("Relaxing Durability," p. 56) can be traded off. If you've heard anything about NoSQL, you'll almost certainly have heard of the CAP theorem; the "The CAP Theorem" section on p. 53 explains what it is and how it fits in.

While these chapters concentrate primarily on the principles of how data gets distributed and kept consistent, the next two chapters talk about a couple of important tools that make this work. Chapter 6 describes version stamps, which are for keeping track of changes and detecting inconsistencies. Chapter 7 outlines map-reduce, which is a particular way of organizing parallel computation that fits in well with clusters and thus with NoSQL systems.

Once we're done with concepts, we move to implementation issues by looking at some example databases under the four key categories: Chapter 8 uses Riak as an example of key-value databases, Chapter 9 takes MongoDB as an example for document databases, Chapter 10 chooses Cassandra to explore column-family databases, and finally Chapter 11 plucks Neo4J as an example of graph databases. We must stress that this is not a comprehensive study—there are too many out there to write about, let alone for us to try. Nor does our choice of examples imply any recommendations. Our aim here is to give you a feel for the variety of stores that exist and for how different database technologies use the concepts we outlined earlier. You'll see what kind of code you need to write to program against these systems and get a glimpse of the mindset you'll need to use them.

A common statement about NoSQL databases is that since they have no schema, there is no difficulty in changing the structure of data during the life of an application. We disagree—a schemaless database still has an implicit schema that needs change discipline when you implement it, so Chapter 12 explains how to do data migration both for strong schemas and for schemaless systems.

All of this should make it clear that NoSQL is not a single thing, nor is it something that will replace relational databases. Chapter 13 looks at this future world of Polyglot Persistence, where multiple data-storage worlds coexist, even within the same application. Chapter 14 then expands our horizons beyond this book, considering other technologies that we haven't covered that may also be a part of this polyglot-persistent world.

With all of this information, you are finally at a point where you can make a choice of what data storage technologies to use, so our final chapter ("Choosing Your Database," p. 147) offers some advice on how to think about these choices. In our view, there are two key factors—finding a productive programming model where the data storage model is well aligned to your application, and ensuring that you can get the data access performance and resilience you need. Since this is early days in the NoSQL life story, we're afraid that we don't have a well-defined procedure to follow, and you'll need to test your options in the context of your needs.

This is a brief overview—we've been very deliberate in limiting the size of this book. We've selected the information we think is the most important—so that you don't have to. If you are going to seriously investigate these technologies, you'll need to go further than what we cover here, but we hope this book provides a good context to start you on your way.

We also need to stress that this is a very volatile field of the computer industry. Important aspects of these stores are changing every year—new features, new databases. We've made a strong effort to focus on concepts, which we think will be valuable to understand even as the underlying technology changes. We're pretty confident that most of what we say will have this longevity, but absolutely sure that not all of it will.

Who Should Read This Book

Our target audience for this book is people who are considering using some form of a NoSQL database. This may be for a new project, or because they are hitting barriers that are suggesting a shift on an existing project.

Our aim is to give you enough information to know whether NoSQL technology makes sense for your needs, and if so which tool to explore in more depth. Our primary imagined audience is an architect or technical lead, but we think this book is also valuable for people involved in software management who want to get an overview of this new technology. We also think that if you're a developer who wants an overview of this technology, this book will be a good starting point.

We don't go into the details of programming and deploying specific databases here—we leave that for specialist books. We've also been very firm on a page limit, to keep this book a brief introduction. This is the kind of book we think you should be able to read on a plane flight: It won't answer all your questions but should give you a good set of questions to ask.

If you've already delved into the world of NoSQL, this book probably won't commit any new items to your store of knowledge. However, it may still be useful by helping you explain what you've learned to others. Making sense of the issues around NoSQL is important—particularly if you're trying to persuade someone to consider using NoSQL in a project.

What Are the Databases

In this book, we've followed a common approach of categorizing NoSQL databases according to their data model. Here is a table of the four data models and some of the databases that fit each model. This is not a comprehensive list—it only mentions the more common databases we've come across. At the time of writing, you can find more comprehensive lists at http://nosql-database.org and http://nosql.mypopescu.com/kb/nosql. For each category, we mark with italics the database we use as an example in the relevant chapter.

Our goal is to pick a representative tool from each of the categories of the databases. While we talk about specific examples, most of the discussion should apply to the entire category, even though these products are unique and cannot be generalized as such. We will pick one database for each of the key-value, document, column family, and graph databases; where appropriate, we will mention other products that may fulfill a specific feature need.

Data Model	Example Databases
Key-Value ("Key-Value Databases," p. 81)	BerkeleyDB
	LevelDB
	Memcached
	Project Voldemort
	Redis
	Riak
Document ("Document Databases," p. 89)	CouchDB
	MongoDB
	OrientDB
	RavenDB
	Terrastore
Column-Family ("Column-Family Stores," p. 99)	Amazon SimpleDB
	Cassandra
	HBase
	Hypertable
Graph ("Graph Databases," p. 111)	FlockDB
	HyperGraphDB
	Infinite Graph
	Neo4J
	OrientDB

This classification by data model is useful, but crude. The lines between the different data models, such as the distinction between key-value and document databases ("Key-Value and Document Data Models," p. 20), are often blurry. Many databases don't fit cleanly into categories; for example, OrientDB calls itself both a document database and a graph database.

Acknowledgments

Our first thanks go to our colleagues at ThoughtWorks, many of whom have been applying NoSQL to our delivery projects over the last couple of years. Their experiences have been a primary source both of our motivation in writing this book and of practical information on the value of this technology. The positive experience we've had so far with NoSQL data stores is the basis of our view that this is an important technology and a significant shift in data storage.

We'd also like to thank various groups who have given public talks, published articles, and blogs on their use of NoSQL. Much progress in software development gets hidden when people don't share with their peers what they've learned. Particular thanks here go to Google and Amazon whose papers on Bigtable and Dynamo were very influential in getting the NoSQL movement going. We also thank companies that have sponsored and contributed to the open-source development of NoSQL databases. An interesting difference with previous shifts in data storage is the degree to which the NoSQL movement is rooted in open-source work.

Particular thanks go to ThoughtWorks for giving us the time to work on this book. We joined ThoughtWorks at around the same time and have been here for over a decade. ThoughtWorks continues to be a very hospitable home for us, a source of knowledge and practice, and a welcome environment of openly sharing what we learn—so different from the traditional systems delivery organizations.

Bethany Anders-Beck, Ilias Bartolini, Tim Berglund, Duncan Craig, Paul Duvall, Oren Eini, Perryn Fowler, Michael Hunger, Eric Kascic, Joshua Kerievsky, Anand Krishnaswamy, Bobby Norton, Ade Oshineye, Thiyagu Palanisamy, Prasanna Pendse, Dan Pritchett, David Rice, Mike Roberts, Marko Rodriquez, Andrew Slocum, Toby Tripp, Steve Vinoski, Dean Wampler, Jim Webber, and Wee Witthawaskul reviewed early drafts of this book and helped us improve it with their advice.

Additionally, Pramod would like to thank Schaumburg Library for providing great service and quiet space for writing; Arhana and Arula, my beautiful daughters, for their understanding that daddy would go to the library and not take them along; Rupali, my beloved wife, for her immense support and help in keeping me focused.

This page intentionally left blank

Part I

Understand

This page intentionally left blank

Chapter 1

Why NoSQL?

For almost as long as we've been in the software profession, relational databases have been the default choice for serious data storage, especially in the world of enterprise applications. If you're an architect starting a new project, your only choice is likely to be which relational database to use. (And often not even that, if your company has a dominant vendor.) There have been times when a database technology threatened to take a piece of the action, such as object databases in the 1990's, but these alternatives never got anywhere.

After such a long period of dominance, the current excitement about NoSQL databases comes as a surprise. In this chapter we'll explore why relational databases became so dominant, and why we think the current rise of NoSQL databases isn't a flash in the pan.

1.1 The Value of Relational Databases

Relational databases have become such an embedded part of our computing culture that it's easy to take them for granted. It's therefore useful to revisit the benefits they provide.

1.1.1 Getting at Persistent Data

Probably the most obvious value of a database is keeping large amounts of persistent data. Most computer architectures have the notion of two areas of memory: a fast volatile "main memory" and a larger but slower "backing store." Main memory is both limited in space and loses all data when you lose power or something bad happens to the operating system. Therefore, to keep data around, we write it to a backing store, commonly seen a disk (although these days that disk can be persistent memory).

The backing store can be organized in all sorts of ways. For many productivity applications (such as word processors), it's a file in the file system of the operating system. For most enterprise applications, however, the backing store is a database. The database allows more flexibility than a file system in storing large amounts of data in a way that allows an application program to get at small bits of that information quickly and easily.

1.1.2 Concurrency

Enterprise applications tend to have many people looking at the same body of data at once, possibly modifying that data. Most of the time they are working on different areas of that data, but occasionally they operate on the same bit of data. As a result, we have to worry about coordinating these interactions to avoid such things as double booking of hotel rooms.

Concurrency is notoriously difficult to get right, with all sorts of errors that can trap even the most careful programmers. Since enterprise applications can have lots of users and other systems all working concurrently, there's a lot of room for bad things to happen. Relational databases help handle this by controlling all access to their data through transactions. While this isn't a cure-all (you still have to handle a transactional error when you try to book a room that's just gone), the transactional mechanism has worked well to contain the complexity of concurrency.

Transactions also play a role in error handling. With transactions, you can make a change, and if an error occurs during the processing of the change you can roll back the transaction to clean things up.

1.1.3 Integration

Enterprise applications live in a rich ecosystem that requires multiple applications, written by different teams, to collaborate in order to get things done. This kind of inter-application collaboration is awkward because it means pushing the human organizational boundaries. Applications often need to use the same data and updates made through one application have to be visible to others.

A common way to do this is **shared database integration** [Hohpe and Woolf] where multiple applications store their data in a single database. Using a single database allows all the applications to use each others' data easily, while the database's concurrency control handles multiple applications in the same way as it handles multiple users in a single application.

1.1.4 A (Mostly) Standard Model

Relational databases have succeeded because they provide the core benefits we outlined earlier in a (mostly) standard way. As a result, developers and database professionals can learn the basic relational model and apply it in many projects. Although there are differences between different relational databases, the core mechanisms remain the same: Different vendors' SQL dialects are similar, transactions operate in mostly the same way.

1.2 Impedance Mismatch

Relational databases provide many advantages, but they are by no means perfect. Even from their early days, there have been lots of frustrations with them.

For application developers, the biggest frustration has been what's commonly called the **impedance mismatch**: the difference between the relational model and the in-memory data structures. The relational data model organizes data into a structure of tables and rows, or more properly, relations and tuples. In the relational model, a **tuple** is a set of name-value pairs and a **relation** is a set of tuples. (The relational definition of a tuple is slightly different from that in mathematics and many programming languages with a tuple data type, where a tuple is a sequence of values.) All operations in SQL consume and return relations, which leads to the mathematically elegant relational algebra.

This foundation on relations provides a certain elegance and simplicity, but it also introduces limitations. In particular, the values in a relational tuple have to be simple—they cannot contain any structure, such as a nested record or a list. This limitation isn't true for in-memory data structures, which can take on much richer structures than relations. As a result, if you want to use a richer in-memory data structure, you have to translate it to a relational representation to store it on disk. Hence the impedance mismatch—two different representations that require translation (see Figure 1.1).

The impedance mismatch is a major source of frustration to application developers, and in the 1990s many people believed that it would lead to relational databases being replaced with databases that replicate the in-memory data structures to disk. That decade was marked with the growth of object-oriented programming languages, and with them came object-oriented databases—both looking to be the dominant environment for software development in the new millennium.

However, while object-oriented languages succeeded in becoming the major force in programming, object-oriented databases faded into obscurity. Relational databases saw off the challenge by stressing their role as an integration mechanism, supported by a mostly standard language of data manipulation (SQL) and a growing professional divide between application developers and database administrators.

Impedance mismatch has been made much easier to deal with by the wide availability of object-relational mapping frameworks, such as Hibernate and iBATIS that implement well-known mapping patterns [Fowler PoEAA], but the mapping problem is still an issue. Object-relational mapping frameworks remove

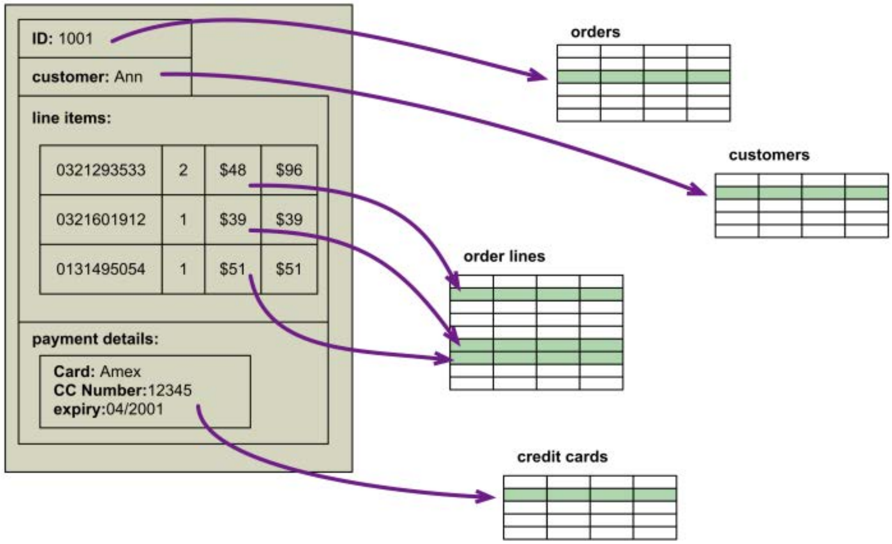


Figure 1.1 *An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database*

a lot of grunt work, but can become a problem of their own when people try too hard to ignore the database and query performance suffers.

Relational databases continued to dominate the enterprise computing world in the 2000s, but during that decade cracks began to open in their dominance.

1.3 Application and Integration Databases

The exact reasons why relational databases triumphed over OO databases are still the subject of an occasional pub debate for developers of a certain age. But in our view, the primary factor was the role of SQL as an integration mechanism between applications. In this scenario, the database acts as an **integration database**—with multiple applications, usually developed by separate teams, storing their data in a common database. This improves communication because all the applications are operating on a consistent set of persistent data.

There are downsides to shared database integration. A structure that's designed to integrate many applications ends up being more complex—indeed, often dramatically more complex—than any single application needs. Furthermore, should an application want to make changes to its data storage, it needs to coordinate with all the other applications using the database. Different applications have different structural and performance needs, so an index required by one application may cause a problematic hit on inserts for another. The fact that each application is usually a separate team also means that the database usually cannot trust applications to update the data in a way that preserves database integrity and thus needs to take responsibility for that within the database itself.

A different approach is to treat your database as an **application database**—which is only directly accessed by a single application codebase that's looked after by a single team. With an application database, only the team using the application needs to know about the database structure, which makes it much easier to maintain and evolve the schema. Since the application team controls both the database and the application code, the responsibility for database integrity can be put in the application code.

Interoperability concerns can now shift to the interfaces of the application, allowing for better interaction protocols and providing support for changing them. During the 2000s we saw a distinct shift to web services [Daigneau], where applications would communicate over HTTP. Web services enabled a new form of a widely used communication mechanism—a challenger to using the SQL with shared databases. (Much of this work was done under the banner of "Service-Oriented Architecture"—a term most notable for its lack of a consistent meaning.)

An interesting aspect of this shift to web services as an integration mechanism was that it resulted in more flexibility for the structure of the data that was being exchanged. If you communicate with SQL, the data must be structured as relations. However, with a service, you are able to use richer data structures with nested records and lists. These are usually represented as documents in XML or, more recently, JSON. In general, with remote communication you want to reduce the number of round trips involved in the interaction, so it's useful to be able to put a rich structure of information into a single request or response.

If you are going to use services for integration, most of the time web services—using text over HTTP—is the way to go. However, if you are dealing with highly performance-sensitive interactions, you may need a binary protocol. Only do this if you are sure you have the need, as text protocols are easier to work with—consider the example of the Internet.

Once you have made the decision to use an application database, you get more freedom of choosing a database. Since there is a decoupling between your internal database and the services with which you talk to the outside world, the outside world doesn't have to care how you store your data, allowing you to consider nonrelational options. Furthermore, there are many features of relational databases, such as security, that are less useful to an application database because they can be done by the enclosing application instead.

Despite this freedom, however, it wasn't apparent that application databases led to a big rush to alternative data stores. Most teams that embraced the application database approach stuck with relational databases. After all, using an application database yields many advantages even ignoring the database flexibility (which is why we generally recommend it). Relational databases are familiar and usually work very well or, at least, well enough. Perhaps, given time, we might have seen the shift to application databases to open a real crack in the relational hegemony—but such cracks came from another source.

1.4 Attack of the Clusters

At the beginning of the new millennium the technology world was hit by the busting of the 1990s dot-com bubble. While this saw many people questioning the economic future of the Internet, the 2000s did see several large web properties dramatically increase in scale.

This increase in scale was happening along many dimensions. Websites started tracking activity and structure in a very detailed way. Large sets of data appeared: links, social networks, activity in logs, mapping data. With this growth in data came a growth in users—as the biggest websites grew to be vast estates regularly serving huge numbers of visitors.

Coping with the increase in data and traffic required more computing resources. To handle this kind of increase, you have two choices: up or out. Scaling up implies bigger machines, more processors, disk storage, and memory. But bigger machines get more and more expensive, not to mention that there are real limits as your size increases. The alternative is to use lots of small machines in a cluster. A cluster of small machines can use commodity hardware and ends up being cheaper at these kinds of scales. It can also be more resilient—while individual machine failures are common, the overall cluster can be built to keep going despite such failures, providing high reliability.

As large properties moved towards clusters, that revealed a new problem—relational databases are not designed to be run on clusters. Clustered relational databases, such as the Oracle RAC Server, work on the concept of a shared disk subsystem. They use a cluster-aware file system that writes to a highly available disk subsystem—but this means the cluster still has the disk subsystem as a single point of failure. Relational databases could also be run as separate servers for different sets of data, effectively sharding ("Sharding," p. 38) the database. While this separates the load, all the sharding has to be controlled by the application which has to keep track of which database server to talk to for each bit of data. Also, we lose any querying, referential integrity, transactions, or consistency controls that cross shards. A phrase we often hear in this context from people who've done this is "unnatural acts."

These technical issues are exacerbated by licensing costs. Commercial relational databases are usually priced on a single-server assumption, so running on a cluster raised prices and led to frustrating negotiations with purchasing departments.

This mismatch between relational databases and clusters led some organization to consider an alternative route to data storage. Two companies in particular—Google and Amazon—have been very influential. Both were on the forefront of running large clusters of this kind; furthermore, they were capturing huge amounts of data. These things gave them the motive. Both were successful and growing companies with strong technical components, which gave them the means and opportunity. It was no wonder they had murder in mind for their relational databases. As the 2000s drew on, both companies produced brief but highly influential papers about their efforts: BigTable from Google and Dynamo from Amazon.

It's often said that Amazon and Google operate at scales far removed from most organizations, so the solutions they needed may not be relevant to an average organization. While it's true that most software projects don't need that level of scale, it's also true that more and more organizations are beginning to explore what they can do by capturing and processing more data—and to run into the same problems. So, as more information leaked out about what Google and Amazon had done, people began to explore making databases along similar lines—explicitly designed to live in a world of clusters. While the earlier menaces to relational dominance turned out to be phantoms, the threat from clusters was serious.

1.5 The Emergence of NoSQL

It's a wonderful irony that the term "NoSQL" first made its appearance in the late 90s as the name of an open-source relational database [Strozzi NoSQL]. Led by Carlo Strozzi, this database stores its tables as ASCII files, each tuple represented by a line with fields separated by tabs. The name comes from the fact that the database doesn't use SQL as a query language. Instead, the database is manipulated through shell scripts that can be combined into the usual UNIX pipelines. Other than the terminological coincidence, Strozzi's NoSQL had no influence on the databases we describe in this book.

The usage of "NoSQL" that we recognize today traces back to a meetup on June 11, 2009 in San Francisco organized by Johan Oskarsson, a software developer based in London. The example of BigTable and Dynamo had inspired a bunch of projects experimenting with alternative data storage, and discussions of these had become a feature of the better software conferences around that time. Johan was interested in finding out more about some of these new databases while he was in San Francisco for a Hadoop summit. Since he had little time there, he felt that it wouldn't be feasible to visit them all, so he decided to host a meetup where they could all come together and present their work to whoever was interested.

Johan wanted a name for the meetup—something that would make a good Twitter hashtag: short, memorable, and without too many Google hits so that a search on the name would quickly find the meetup. He asked for suggestions on the #cassandra IRC channel and got a few, selecting the suggestion of "NoSQL" from Eric Evans (a developer at Rackspace, no connection to the DDD Eric

Evans). While it had the disadvantage of being negative and not really describing these systems, it did fit the hashtag criteria. At the time they were thinking of only naming a single meeting and were not expecting it to catch on to name this entire technology trend [Oskarsson].

The term "NoSQL" caught on like wildfire, but it's never been a term that's had much in the way of a strong definition. The original call [NoSQL Meetup] for the meetup asked for "open-source, distributed, nonrelational databases." The talks there [NoSQL Debrief] were from Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, and MongoDB—but the term has never been confined to that original septet. There's no generally accepted definition, nor an authority to provide one, so all we can do is discuss some common characteristics of the databases that tend to be called "NoSQL."

To begin with, there is the obvious point that NoSQL databases don't use SQL. Some of them do have query languages, and it makes sense for them to be similar to SQL in order to make them easier to learn. Cassandra's CQL is like this—"exactly like SQL (except where it's not)" [CQL]. But so far none have implemented anything that would fit even the rather flexible notion of standard SQL. It will be interesting to see what happens if an established NoSQL database decides to implement a reasonably standard SQL; the only predictable outcome for such an eventuality is plenty of argument.

Another important characteristic of these databases is that they are generally open-source projects. Although the term NoSQL is frequently applied to closed-source systems, there's a notion that NoSQL is an open-source phenomenon.

Most NoSQL databases are driven by the need to run on clusters, and this is certainly true of those that were talked about during the initial meetup. This has an effect on their data model as well as their approach to consistency. Relational databases use ACID transactions (p. 19) to handle consistency across the whole database. This inherently clashes with a cluster environment, so NoSQL databases offer a range of options for consistency and distribution.

However, not all NoSQL databases are strongly oriented towards running on clusters. Graph databases are one style of NoSQL databases that uses a distribution model similar to relational databases but offers a different data model that makes it better at handling data with complex relationships.

NoSQL databases are generally based on the needs of the early 21st century web estates, so usually only systems developed during that time frame are called NoSQL—thus ruling out hoards of databases created before the new millennium, let alone BC (Before Codd).

NoSQL databases operate without a schema, allowing you to freely add fields to database records without having to define any changes in structure first. This is particularly useful when dealing with nonuniform data and custom fields which forced relational databases to use names like `customField6` or custom field tables that are awkward to process and understand.

All of the above are common characteristics of things that we see described as NoSQL databases. None of these are definitional, and indeed it's likely that there will never be a coherent definition of "NoSQL" (sigh). However, this crude set of characteristics has been our guide in writing this book. Our chief enthusiasm with this subject is that the rise of NoSQL has opened up the range of options for data storage. Consequently, this opening up shouldn't be confined to what's usually classed as a NoSQL store. We hope that other data storage options will become more acceptable, including many that predate the NoSQL movement. There is a limit, however, to what we can usefully discuss in this book, so we've decided to concentrate on this noDefinition.

When you first hear "NoSQL," an immediate question is what does it stand for—a "no" to SQL? Most people who talk about NoSQL say that it really means "Not Only SQL," but this interpretation has a couple of problems. Most people write "NoSQL" whereas "Not Only SQL" would be written "NOSQL." Also, there wouldn't be much point in calling something a NoSQL database under the "not only" meaning—because then, Oracle or Postgres would fit that definition, we would prove that black equals white and would all get run over on crosswalks.

To resolve this, we suggest that you don't worry about what the term stands for, but rather about what it means (which is recommended with most acronyms). Thus, when "NoSQL" is applied to a database, it refers to an ill-defined set of mostly open-source databases, mostly developed in the early 21st century, and mostly not using SQL.

The "not-only" interpretation does have its value, as it describes the ecosystem that many people think is the future of databases. This is in fact what we consider to be the most important contribution of this way of thinking—it's better to think of NoSQL as a movement rather than a technology. We don't think that relational databases are going away—they are still going to be the most common form of database in use. Even though we've written this book, we still recommend relational databases. Their familiarity, stability, feature set, and available support are compelling arguments for most projects.

The change is that now we see relational databases as one option for data storage. This point of view is often referred to as **polyglot persistence**—using different data stores in different circumstances. Instead of just picking a relational database because everyone does, we need to understand the nature of the data we're storing and how we want to manipulate it. The result is that most organizations will have a mix of data storage technologies for different circumstances.

In order to make this polyglot world work, our view is that organizations also need to shift from integration databases to application databases. Indeed, we assume in this book that you'll be using a NoSQL database as an application database; we don't generally consider NoSQL databases a good choice for integration databases. We don't see this as a disadvantage as we think that even if you don't use NoSQL, shifting to encapsulating data in services is a good direction to take.

In our account of the history of NoSQL development, we've concentrated on big data running on clusters. While we think this is the key thing that drove the opening up of the database world, it isn't the only reason we see project teams considering NoSQL databases. An equally important reason is the old frustration with the impedance mismatch problem. The big data concerns have created an opportunity for people to think freshly about their data storage needs, and some development teams see that using a NoSQL database can help their productivity by simplifying their database access even if they have no need to scale beyond a single machine.

So, as you read the rest of this book, remember there are two primary reasons for considering NoSQL. One is to handle data access with sizes and performance that demand a cluster; the other is to improve the productivity of application development by using a more convenient data interaction style.

1.6 Key Points

- Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism.
- Application developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures.
- There is a movement away from using databases as integration points towards encapsulating databases within applications and integrating through services.
- The vital factor for a change in data storage was the need to support large volumes of data by running on clusters. Relational databases are not designed to run efficiently on clusters.
- NoSQL is an accidental neologism. There is no prescriptive definition—all you can make is an observation of common characteristics.
- The common characteristics of NoSQL databases are
    - Not using the relational model
    - Running well on clusters
    - Open-source
    - Built for the 21st century web estates
    - Schemaless
- The most important result of the rise of NoSQL is Polyglot Persistence.

Index

D

Q

R