Making Everything Easier!™

MarkLogic Special Edition

Enterprise NoSQL for Dummies®

A Wiley Brand

Learn to:

- **Do more with your structured and unstructured data**
- **Combine disparate data sources without the limits of relational systems**
- **Build applications faster within an agile data architecture**

Brought to you by

Charlie Brooks

Since MarkLogic was founded in 2001, the company has focused on building a database platform that enables customers to capture more data and do more with it. Customers gain an unmatched competitive edge through game-changing technology that sets new standards in scalability, enterprise-readiness, time-to-value, and innovation.

The world is seeing an explosion of data, including user-generated content, machine-generated data, highly-structured heterogeneous data sources (patient records, insurance claims, mortgage documents), raw data, social media, log files, sensor data, and more. Relational databases weren't designed—and simply are not able—to deal with this variety and complexity in real-time. This is exactly the problem MarkLogic has been solving for over a decade.

MarkLogic customers are solving challenges with Big Data applications that were never imagined before. The Federal Aviation Administration depends on MarkLogic as the backbone of its Emergency Operations Network. Royal Society of Chemistry is building new applications in weeks. CQ Roll Call puts up-to-date information about government-in-action at the fingertips of its subscribers.

MarkLogic is headquartered in Silicon Valley with offices in Washington D.C., New York, Chicago, London, Frankfurt, Utrecht, and Tokyo.

For more information on MarkLogic, visit **www.marklogic.com.**

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Enterprise NoSQL

FOR

DUMMIES®

A Wiley Brand

MarkLogic Special Edition

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Enterprise NoSQL FOR DUMMIES® A Wiley Brand

MarkLogic Special Edition

by Charlie Brooks

FOR DUMMIES® A Wiley Brand

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Enterprise NoSQL For Dummies®, MarkLogic Special Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

Copyright © 2014 by John Wiley & Sons, Inc.

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, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Trademarks: Wiley, For Dummies, the Dummies Man logo, The Dummies Way, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries, and may not be used without written permission. MarkLogic and the MarkLogic logo are registered trademarks of MarkLogic. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, or how to create a custom *For Dummies* book for your business or organization, please contact our Business Development Department in the U.S. at 877-409-4177, contact info@dummies.biz, or visit www.wiley.com/go/custompub. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

ISBN: 978-1-118-82900-4 (pbk); ISBN: 978-1-118-83261-5 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Table of Contents

Introduction 1

About This Book 1
Icons Used in This Book 2
Beyond the Book 2

Chapter 1: NoSQL 101 5

Talking the NoSQL Talk 5
What NoSQL Is 7
Four database flavors 7
Key-value 7
Document 8
Column-family 8
Graph 8
Schema-agnostic 10
Free of complex joins 10
Horizontally scaleable 10
Compatible with commodity hardware 11
Self-contained 11
What NoSQL Isn't 12
Just about the SQL query language 12
Only for huge enterprises 12
A drop-in RDBMS replacement 13
Dependent on specialized hardware 13
Intolerant of relational databases 14
Is NoSQL Right for Your Business? 14

Chapter 2: What's the Difference in DBMSes? 17

Comparing RDBMS and NoSQL 17
RDBMS fundamentals 18
NoSQL alternatives 19
Working on Your Relationships 20
RDBMS design 21
NoSQL design 22
Getting Back to Denormal 23

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 3: Going Beyond the Data Center with NoSQL 25

Data Distribution........ 25
Sharding........ 26
Replication........ 27
Master/slave replication........ 27
Peer-to-peer replication........ 27
The ACID Test: Data Consistency........ 28
ACID constraints in NoSQL........ 29
Read and update consistency........ 29
Document Durability........ 30

Chapter 4: Seeing What Enterprise NoSQL Can Do . . .33

Enterprise Replication........ 34
Enterprise Data Backups........ 35
Multiple-Record ACID Transactions........ 36
Enterprise-Class Security........ 37
Authorization and accounting........ 37
Authentication........ 37
Access control........ 38
Compartment security........ 38
Policy Management........ 39
Administration and Management Tools........ 40
High Availability........ 41
Scalability........ 42
Integration with Third-Party Solutions........ 42

Chapter 5: Real-World Enterprise NoSQL. 43

Consolidating Disparate Data........ 43
County government........ 43
Problem........ 44
Solution........ 45
Benefits........ 45
Investment banking........ 46
Sharpening Situational Awareness........ 46
Storing Operational Data........ 48
Discovering and Repurposing Data........ 49
Publishing........ 51
Broadcasting........ 51

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 6: Ten Questions to Ask About NoSQL Solutions 53

Are You Ready to Incorporate the Solution?........................ 53
Is the Solution Agile Enough? 54
Does the Solution Meet Your Application Requirements?.. 55
How Much Development Is Required to Get Data In? 55
Do You Need New Tools?.. 56
Does the Solution Support Your Cloud Strategy?................ 56
Is the Solution Enterprise-Ready?.. 57
Do You Need Specialized Hardware? 57
Will the Solution Grow with Your Business?........................ 58
Can the Solution Speed Application Development? 58

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Publisher's Acknowledgments

We're proud of this book and of the people who worked on it. For details on how to create a custom *For Dummies* book for your business or organization, contact `info@dummies.biz` or visit `www.wiley.com/go/custompub`. For details on licensing the *For Dummies* brand for products or services, contact `BrandedRights&Licenses@Wiley.com`.

Some of the people who helped bring this book to market include the following:

Project Editor: Carrie A. Johnson

Acquisitions Editor: Connie Santisteban

Editorial Manager: Rev Mengle

Business Development Representative: Karen Hattan

Custom Publishing Project Specialist: Michael Sullivan

Production Coordinator: Melissa Cossell

Special Help: Adam Fowler, Blossom Coryat, and Norm Walsh

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Introduction

You've probably heard about NoSQL, and you may wonder what it is. NoSQL represents a fundamental change in the way people think about storing and accessing data, especially now that most of the information generated is unstructured or semi-structured data — something for which existing database systems such as Oracle, MySQL, SQLServer, and Postgres aren't well suited.

NoSQL casts a very wide net (trying to offer a precise definition of NoSQL is like trying to nail gelatin to a tree). Search for *NoSQL databases* on the Internet, and you're likely to find some common themes, such as non-relational, horizontally scalable, and (mostly) schema-agnostic. But what does that mean? NoSQL means a release from the constraints imposed on database management systems by the relational database model.

About This Book

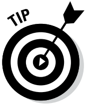
Enterprise NoSQL For Dummies, MarkLogic Special Edition, provides an overview of NoSQL. You start to understand what it is, what it isn't, when you should consider using a NoSQL database instead of a relational database management system (RDBMS) and when you may want to use both. In addition, this book introduces enterprise NoSQL and shows how it differs from other NoSQL systems.

NoSQL may not be a solution for every data storage problem. Also, not all NoSQL databases are created equal, or even equel. (Sorry — a little database humor there.) Though I certainly hope that you read the book cover to cover, the book is organized in such a way that you can sample the chapters that you find most interesting and useful in whatever order you want.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Icons Used in This Book

Every once in a while in the body of the book, you see an icon in the left margin. Each icon indicates a particular paragraph or two that you should pay attention to.



Tips help save time or minimize frustration about a particular product or procedure. You could save more or learn a new way of doing something.

Remember icons represent information that you file away in your brain, even if you remember nothing else.

Warning icons represent serious situations involving risk to your work products or your organization's overall operations. Heed the information here to avoid some pitfalls of NoSQL.

Technical Stuff is information that you don't need to know unless you want a deeper understanding of what's under the hood. Feel free to skip these paragraphs, but come back to them later if you want to know more technical information about how NoSQL works.

Beyond the Book

While you're reading this book, you may be thinking, "Are there really databases that can do x, y, or z?" The answer is yes, they can, and those databases do exist.

Then you may think, "But that work may take rafts of developers and architects to configure and to get data into the database." In fact, that's not true either — it's easy to put data into the database in seconds and search that data instantly. You can even get search results from within attachments. It doesn't take eons to connect the data to an application or build a new application.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Enterprise NoSQL For Dummies, MarkLogic Special Edition, can help you discover more about enterprise NoSQL, but if you want resources beyond what this book offers, I have some insight for you:

- MarkLogic is an enterprise grade NoSQL database. It has customers who use its services and are building new applications in weeks and months, instead of years. Check out testimonials here: `http://po.st/tst4d`.
- Try it for yourself. Search for content in one of the over 67 million emails or attachments in MarkMail. Go to `http://markmail.org`.
- Download MarkLogic and see what it can do with your data. Visit `www.marklogic.com`.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 1

NoSQL 101

In This Chapter

- Understanding some key NoSQL terms
- Seeing what NoSQL is — and isn't
- Deciding whether NoSQL can benefit your business

NoSQL isn't truly new (although some of the software certainly is). Folks were defining and using database structures other than a relational database management system (RDBMS) back in the 1980s. What *is* new are the challenges introduced by Big Data.

Big Data is a large and complex collection of data sets that occurs because of the velocity, volume, variety, and complexity of the data. The challenge this creates includes capture, curation, storage, search, sharing, transfer, analysis, and visualization.

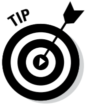
NoSQL isn't so much an advance in technology (aside from the development of commodity computers) as it is a way of matching database performance and capability to application demands and giving users the ability to use data sources they never thought they could access. Using NoSQL requires a change in mindset, moving from the RDBMS model to one designed to best support application queries. This chapter helps prepare you for that change. I cover what NoSQL is about and address some common misconceptions.

Talking the NoSQL Talk

When you start exploring NoSQL, you may be hit by a barrage of new terms or terms used in a way that you don't understand. Fear not, dear reader: This section gets you caught up with

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

NoSQL terminology. Soon, you'll be discussing *sharding* and *replication* with the best of them.



First, though, I'm sure you wonder how to pronounce *NoSQL.* I've heard it pronounced as *no sequel* and as *no S-Q-L.* My advice: Listen to how the people you're interacting with pronounce it, and copy them.

Here are some important NoSQL terms:

- **Node:** A *node* is a networked computer that offers some kind of service (usually a compute service), local storage, and access to a much larger distributed data or file store.
- **Clusters:** As used in NoSQL land, a *cluster* is a set of nodes that constitute a single unit. Depending on the database, a cluster can be a set of nodes on a particular rack in the data center or nodes that are in the same row as other nodes. I talk more about "rack versus row" in Chapter 3.
- **Sharding:** Sharding (also called *horizontal partitioning*) involves partitioning the database on the value of some field. This is done by some NoSQL databases to equalize the amount of data between nodes. You can do sharding directly by hashing the key value or by *load balancing* — directing each node to redistribute its data to another, less heavily loaded node (see Chapter 3).
- **Replication:** *Replication* is the mechanism that provides database availability. Portions of a database are written to multiple nodes so that if one node fails, another node contains a replica of the failed node's data. For details on replication, see Chapter 3.
- **ACID:** *ACID* stands for *atomicity, consistency, isolation,* and *durability.* ACID is a watchword in transactional systems (such as those used in banking and e-commerce) and necessary for any system of record. I discuss ACID in Chapter 3.
- **BASE:** *BASE* stands for *basically available, soft-state, and eventually consistent. Basically available* indicates that the database may not be available 24/7. *Soft state* implies that the state of the database may be inconsistent; if you change your work email address, for example, a friend may not see that information immediately. *Eventually consistent,* however, means that your friend eventually sees your changed email information.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

What NoSQL Is

NoSQL can represent several things in the data management world. In the following sections, I discuss a few of its features.

Four database flavors

NoSQL represents a much different data model from earlier models, such as *hierarchical* (which represents relationships in a strict hierarchy) and *network* (which represents many-to-many relationships by introducing the notion of record sets). An RDBMS represents relationships *(entities)* as tables, with each row *(tuple)* representing a particular entity and each column representing an attribute of that entity. Unlike earlier models, the relational model allows only single-valued attributes. If you want to indicate that a particular entity is related to multiple other entities, you have to create a separate table to express that relationship. You can't nest one tuple within another — a process known as *repeating groups.*

Although NoSQL databases frequently refer to *tables, rows,* and *columns,* those terms don't have the same meaning or restrictions as they do in an RDBMS. Check out Table 1-1 for an explanation of how those terms *are* used in NoSQL.

Table 1-1 RDBMS Terms versus NoSQL Terms

RDBMS Term	*NoSQL Term*
Partition	Shard
Table	Document root element (JSON/XML)
Row	Document/aggregate or record
Column	Element/attribute or field or property

NoSQL comes in four main flavors, which I discuss in the following sections.

Key-value

Key-value databases model data as a search/index key and a value represented as an uninterpreted sequence of bytes. This kind of database has been around for many years. You can quickly and easily read a given record based on its key, but you can't search value data across multiple records.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

One example of a key-value database is one in which the key is the URL of a given web page, but the contents can be expressed as HTML, PHP, JavaScript, or even binary data.

Key-value databases usually require a unique key. They have no implicit ordering, but some implementations allow you to order the database by the value of the key, thereby enabling range searches against the key.

Document

Document (sometimes called *aggregate*) databases are very much like key-value databases, except that the value associated with a key contains structured or semi-structured data, which can be labeled as a document. Unlike in a key-value database, you can query against the structure of the document as well as elements within that structure, and return only portions of the document as the results of the query.

An example of a document-oriented database is a book database in which the key is the book title and the value is book metadata expressed as an XML document.

Column-family

Column-family databases can be the most difficult kind to wrap your head around. One way to think about these databases is that they're very large tables with zillions of rows and zillions of possible columns, but each row actually has a relatively small number of columns compared with the total number possible. Mathematicians recognize this arrangement as a *sparse matrix;* programmers may recognize it as a *hash table* or dictionary mapping a key to a set of key-value pairs.

An example of such a database is one in which the key is a URL and each column represents a revision of the document. One column-family contains metadata about that page; another column-family contains data about when the page was modified, what was changed, the extent of the change, and so on.

Graph

Graph databases are different from the preceding three types (see the preceding sections) in that the relationships among the various entities are the most important things. Nodes

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

in the graph represent a particular kind of entity (person or thing), and the edges between nodes are labeled as particular kinds of relationships. Edges point in one direction, and an edge itself may have attributes. Take a look at Figure 1-1 for an example of this type of database.

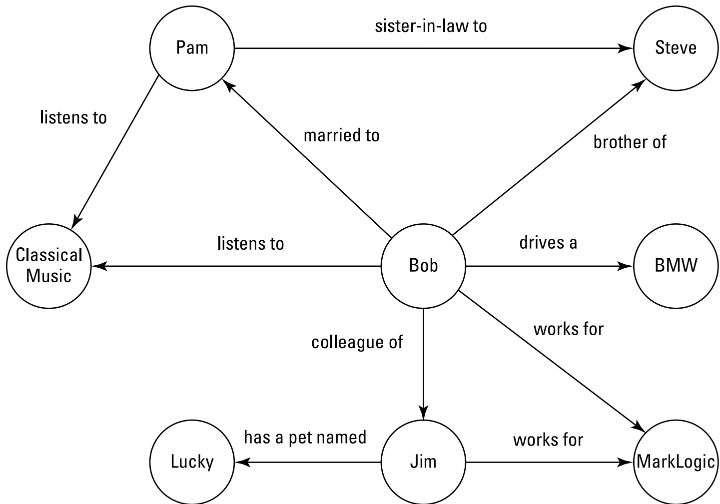


Figure 1-1: A graph database example.

As an example, consider the "following" relationship in the Twitter messaging system. You can represent a relationship between Charlie and Chris by saying that Charlie follows Chris, but that doesn't mean that Chris follows Charlie. You can represent these relationships as pairs, as in

```
Follows (Charlie, Chris)
```

A *triple* is a single piece of information, known as an assertion. An example is

```
Charlie, follows, Chris
```

Over time, grouping these individual assertions together builds a web of facts. This is the technology that the semantic web is built on.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Triple your fun

Here are a few other examples of triples:

```
Paul   worksFor   marklogic

Alex   worksIn.
   Marketing

London  isIn   England

Oracle   isA.
   Relational Database

Document databases  Are
   awesome

Horse Riding  typeOf
   sport
```

Schema-agnostic

Given that a NoSQL database has no tables, rows, or columns in the same sense as these elements are used in relational databases, you shouldn't be surprised to discover that NoSQL databases don't have a notion of a fixed schema, either.

Free of complex joins

NoSQL databases are designed so joins aren't necessary or even recommended. If for some reason they're needed, choose a database, such as MarkLogic, that can perform complex joins.

Horizontally scaleable

As you make more and more demands of your database, you eventually run out of a particular resource, whether it be processing power, storage space, input/output (I/O) cycles, or whatever. At that time, you have a choice to make: Scale up (vertically) by buying later and more powerful hardware, or scale out (horizontally) by buying a new machine of comparable power.

NoSQL favors the latter approach, and NoSQL databases make the job of incorporating new machines into their database clusters easy.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Compatible with commodity hardware

In part because of the design goal of supporting horizontal scalability (see the preceding section), most NoSQL database systems run on commodity hardware. Commodity hardware isn't always inexpensive, however. Recommendations for nodes include lots of memory (16GB or more), multiple CPUs, and a lot of disk space.

One recommendation for a particular machine for a cluster was two quad-core CPUs (2 to 2.5 GHz), ECC RAM (16GB to 24GB), four 10x 600GB SATA disks, and a 1GB Ethernet card. Even today, that machine would cost more than $2,000. On the flip side, a single machine with 10 times the power would cost much more than 10 machines at $2,000 each.

Self-contained

Many database systems share resources, including storage, processor resources, and memory. The NoSQL philosophy, however, is that nothing is shared. Each node is stand-alone with respect to storage and processing power, so queries can run on a particular node without being blocked by another query that locked some portion of the database for its own purposes.

That said, it requires good design and good engineering to provide careful sharding of the database (see Chapter 3) so that queries can run in parallel without interference. This model of passing queries to all nodes and then combining the answer is similar to the MapReduce technique, with the distinction that NoSQL systems return a live operational result, whereas MapReduce systems such as Hadoop operate only in batch mode. Some NoSQL databases provide record sharding automatically, but not all of them.

Not all NoSQL databases run across multiple machines. Graph databases, for example, usually are restricted to a single machine because graph math across nodes is slow.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

What NoSQL Isn't

Even though NoSQL arrived relatively late on the database scene, enough time has passed for various misconceptions to flourish. When you advocate for a NoSQL solution, you may hear questions such as "Well, isn't NoSQL only . . . ?" or "But doesn't NoSQL just do . . . ?" This section gives you good answers to those types of questions.

Just about the SQL query language

Whether you pronounce *NoSQL* as *no SQL* or *not only SQL* (see "Talking the NoSQL Talk" earlier in this chapter), NoSQL isn't concerned about the presence or absence of the SQL query language. Rather, NoSQL is concerned about whether and when the guarantees and restrictions of the relational database model (see Chapter 2) are necessary.

In many cases, NoSQL doesn't offer stand-alone query languages. Instead, the database is accessed via an application programming interface (API).

Some NoSQL databases don't allow string-based queries (such as `Select * from Orders where itemname = "pillows"`). The Hive query language for Hadoop uses a decidedly SQL-like syntax, and the Cassandra database provides the CQL (pronounced *C-quell* — get it?) query language, which looks a lot like SQL.

Only for huge enterprises

Many large companies associated with the cloud (I bet you wondered how long I could go without using that word) use NoSQL databases because they depend on a wide variety of complex, fast-moving data. Yahoo!, Google, and Amazon, for example, all started developing a non-RDBMS in the early 2000s to meet their business needs. These business needs are characterized by Big Data: a lot of variable, constantly

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

changing data, with query results required very quickly. (After all, slow response means that customers will leave for other businesses.)

Yet NoSQL isn't just for large companies. Even small companies may need NoSQL because they're trying to consolidate their data and remove data silos for structured and unstructured data. Your business may operate on a much smaller scale, but you don't need to have petabytes of data to have a Big Data problem. Your Big Data problem could be how to extract patterns and relationships from unstructured or semi-structured data to gain relevant, actionable information about your business, as well as how to access that information quickly so your business can adapt to rapidly changing conditions.

A drop-in RDBMS replacement

A NoSQL DBMS isn't a drop-in replacement for your existing RDBMS. An RDBMS has been developed since the 1970s, and includes many facilities to ensure data is kept safe, including ACID guarantees and indexing for performance.

If you're working with a system of record in an online transaction processing environment with hundreds of transactions per second, for example, there's no reason to rip out that system and start afresh. The truth is, NoSQL is a disruptive technology, not only because of its different perspective on data storage and management, but also because it requires a different skill set for database specialists and application programmers.

Dependent on specialized hardware

NoSQL doesn't require specialized hardware to perform well unlike an RDBMS that uses vertical scaling. At some point, vertical scaling (adding bigger and bigger iron) requires certain functions to be offloaded to special-purpose devices, which increase complexity, power consumption, physical floor space, and so on. With NoSQL, you add more commodity

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

servers, making managing expansion easier with NoSQL instead of an RDBMS. Also many NoSQL solutions can run in the cloud, further improving elasticity options of increasing and decreasing the number of servers at short notice.

Intolerant of relational databases

Choosing a NoSQL database isn't a matter of throwing out the RDBMS baby with the NoSQL bathwater. Using a relational database in tandem with a NoSQL database can be tricky, but it can make business sense. This is true even if the NoSQL database is updated via an ETL (Extract, Transform, and Load) process that runs periodically or in parallel with queries against that database.

One example of combining the two types of databases is using company information or reference data on exchange rates to enrich trading databased as XML documents in the NoSQL database.

Is NoSQL Right for Your Business?

If you've been reading this chapter from start to finish, you may think that all the preceding information is nice, but you may still wonder how NoSQL could make a difference in your business. This section gives you the lowdown on how.

You may be a candidate for NoSQL if

- You have a lot of unstructured or semi-structured data, or a mix of unstructured and relational data.
- You need to support multiple queries while simultaneously loading a lot of data.
- You need to reuse portions of your data for multiple projects.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

- You have rapidly changing schemas or need to take on new information sources without a six-month (or longer) development cycle.
- You need to consolidate multiple, disparate data types and sources without being forced to model data or create a schema.
- You don't know what queries you want to be able to ask of your data.
- You need to be able to search for the contents of a document.

If these descriptions fit your business, a NoSQL solution may be right for you.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 2

What's the Difference in DBMSes?

In This Chapter

- Differentiating between relational and NoSQL data modeling
- Handling relationships
- Understanding (de)normalization

Taking full advantage of any product or solution requires preparation and the proper mindset. To get the most out of it, you have to change the way you think about it. To get the most from a database management system (DBMS), you have to change your mind-set about managing data. "Yeah," you're saying skeptically, "but data is data." Stop right there. If you're about to say something like "A NoSQL database can't be that different from a relational database," you're wrong.

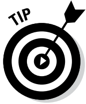
This chapter helps you grasp the differences between a relational database management system (RDBMS) and a NoSQL DBMS.

Comparing RDBMS and NoSQL

What, then, are the key features that differentiate an RDBMS and NoSQL? At one time, IT professionals may have said that an RDBMS emphasizes consistency and a NoSQL DBMS emphasizes availability, but that's no longer the case. The real differences now center on flexibility (schema-agnostic) and methods of handling different data structures.

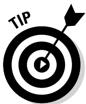
These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

NoSQL databases and relational databases offer different capabilities and guarantees. Just as you won't get the performance you want if you drive a Ferrari in rush-hour traffic, choose one or the other type based on your application requirements.



If you're dealing with structured data that's highly transactional, an RDBMS is probably the way to go. Or you might consider splitting database management, with an RDBMS managing the structured data and a NoSQL solution managing the unstructured data. The latter type of system, called *polyglot persistence,* will be a key solution until NoSQL databases support multiple data structures (as MarkLogic is doing for documents, values, and triple structures; see Chapter 1).

Design your data management architecture and systems to play to the strengths of both types of solutions. You probably don't want to limit your choice of NoSQL solutions to only one model. After all, triple stores and document databases serve different purposes, and using the right tool for the right job still applies.



MarkLogic provides both triple stores and document databases, so if you don't want to limit your choices, check into MarkLogic's solution.

RDBMS fundamentals

Anyone who works with an RDBMS is familiar with the table as a basic way of handling RDBMS data. Here's the structure:

- Rows represent records, and columns represent record attributes or fields.
- Every row in the same table has exactly the same columns, although some columns may not always have a value (and, therefore, are called *null valued)*.
- Each row of a record is identified internally by a special key, even if that key isn't visible to developers.
- Each row in the table usually is unique, identified externally by a primary key (often, a specific combination of fields or a single field).
- An index usually is created on the key and any common lookup field to allow faster lookups.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

- Each field in the table has a single value. Field names must be unique as well. There are no repeating groups or hash maps within a table.

So far, so routine.

The whole purpose of designing the database this way is to prevent update anomalies. The principle is "One fact in one place." You have to access only one record to change a particular value. If you want to change your phone number from 1-413-555-1212 to 1-617-444-1212, for example, you should be able to make that change in one field of one specific record — not require that all data in a record be resubmitted just to update a single field.

One consequence of this approach to designing RDBMS tables is that problems crop up as soon as you need to express relationships among the entities, as you see in "Working on Your Relationships" later in this chapter.

NoSQL alternatives

SQL changed the RDBMS scene by offering organizations an alternative method of storing and managing their data. But why choose yet another alternative — NoSQL — instead? Here are two major reasons:

- **NoSQL represents a practical change in perspective.** Instead of getting the schema just right before doing anything else, NoSQL advocates loading up the data and seeing where the problems are. This problem-oriented approach focuses on how the data will be used (queried) instead of on how the data must be structured to fit an existing RDBMS.
- **NoSQL models data in a way that's more understandable to mere mortals.** Almost anyone can look at a document and make sense out of it. An XML document clearly indicates what each piece of the document is. The RDBMS tables necessary to represent that document are more difficult to understand, and deriving the business rules from the schema is difficult.

As I mention in Chapter 1, a NoSQL is schema-agnostic. You can enforce a schema if you want to, but you usually don't have to.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Keep in mind the following NoSQL facts:

- NoSQL documents don't require each document to have the same attributes.
- NoSQL is explicitly aware of how data will be used in a particular application.
- At the physical database level, NoSQL is aware that it runs in a cluster and takes that fact into account in its data management strategy.
- NoSQL applications can specify a replication factor to ensure a particular level of consistency.

The rest of this chapter provides some concrete examples of the differences between an RDBMS and NoSQL.

Working on Your Relationships

A relationship between database entities may require a separate table to represent that relationship. If you have a *one-to-one (1:1) relationship* between two entities, you can represent this relationship by embedding the key for one entity in the record for the other entity. You can use the same strategy to represent a *one-to-many relationship (1:M).* If you have a *many-to-many relationship (M:M),* you need a separate linking table.

The headaches you have to deal with depend on the type of relationship. Here are a couple of considerations:

- **Referential integrity:** A database constraint called *referential integrity* (data consistency) requires a foreign key, if not `NULL`, to indicate an entity that already exists in the database. But if you use `NULL`, you're indicating that the relationship doesn't exist yet. Therefore, if you want to embed a foreign key in a record, you first have to create the record, using that value as a primary key. That table may have its own referential integrity constraints, however.
- **Mapping tables:** A 1:M relationship may require you to create a mapping table in the database to capture that relationship between these two entities. Otherwise, you can achieve the same results by embedding the key for one entity in the table that represents the other. If you

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

use a mapping table, it can consist of just the keys of each entity, representing that one entity is associated with another. In other cases, this relationship table may have attributes that tell you something more about that relationship.

A M:M relationship, by contrast, requires a mapping table to represent that relationship.

- **Object orientation:** Object-oriented databases that create object-relational mapping between objects and their representations in RDBMS tables don't always work too well. Which is more important: the objects or their relationships? The answer depends on whom you ask and the purpose for which the database is used. In database programming, the object view of the world and the relational view of the world have always been at odds.

Consider an example. Suppose that you need to create a database to capture information about a college: students, professors, courses, classes, classrooms, and the like. Further suppose that each class can have only one professor. In the following sections, I show you how you could design this database for an RDBMS and then how you could create a NoSQL version.

RDBMS design

For this example, your RDBMS table definition may look something like this:

```
create table class {
   section varchar(10) NOT NULL,
   class_id Integer NOT NULL UNIQUE,
   professor : Integer,
   Primary key(class_id),
   Foreign key (professor) references
Professor}
```

This design decision may not be acceptable to everyone. In particular, it can require the use of a value representing NULL in the class record, meaning that a professor hasn't been assigned to this class. Dealing with NULLs complicates life for the application programmer, but sometimes, NULLs are necessary evils. Some NoSQL databases permit document values to be NULL or simply make them not present. I discuss this topic further in the following section.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Empty lists versus null values

Representing a student's classes as an empty list means that you don't have to worry about null values for enrollment. You still face the issue that a student may be enrolled in a class to which no professor has been assigned. In this case, you can use `NULL` to indicate that no professor has been assigned to teach this class.

Which choice is better? Here's a rule of thumb: A null value is better if the document attribute is a single value, whereas an empty list is better if the attribute can have multiple values. A student can be enrolled in 0, 1, or more classes, so the empty list is more appropriate.

As a programmer, you could create a student's record with `s1 = new Student()` and delete the record with `s1.delete()`.This seemingly simple transaction, however, involves making changes in multiple tables and requires the database to manage concurrent access to all these tables to maintain referential integrity across the database.

A table describing a student in an RDBMS might look like this:

```
create table student {
   name : string ,
   student_id : integer NOT NULL,
   year : integer,
   Primary Key (student_id)}
```

To represent that a student can be enrolled in multiple classes, you need to create a new table representing the relationship between a student and a class:

```
create table student_class {
   student_id integer NOT NULL,
   class_id integer NOT NULL,
   primary key (student_id, class_id)}
```

NoSQL design

A NoSQL developer would approach the example design differently, using different names for various database components. In NoSQL, documents don't have to have the same set of attributes and can have attributes that don't exist yet.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Using a NoSQL approach, the student record would look something like this:

```
<student xmlns="http://myeducation.
          com/education" studentId="5"
          year="2014">
<student-name>Joe Bloggs</student-name>
<class classId="CSC101">
<professor professorId="456">
<professor-name>John Adams</professor-name>
<tenured>true</tenured>
<yearsEmployed>23</yearsEmployed>
</professor>
<section>Introduction to Computer Science
          </section>
</class>
<class classId="POL245">
. . .
</class>
</student>
```

Sparse data

Some applications, such as contact lists, have hundreds of fields (home phone, cellphone, email, Twitter ID, and so on), but normally, only a few of these fields are filled out. As a result, many rows of a contact table contain null values. This scenario, called a *sparse-data problem,* results in waste in an RDBMS. NoSQL databases, by contrast, allow you to store or not store information for a particular field, key, or element. Therefore, they don't suffer from the sparse-data problem associated with an RDBMS.

Getting Back to Denormal

Normalizing, in the RDBMS world, is the act of performing a series of transformations on a database design until the design meets certain tests. Each of these tests describes a particular normal form. There are several kinds of normal forms, including 1NF (1st normal form), 2NF, 3NF, 4NF, and BCNF (Boyce-Codd normal form). Depending on your data, the first three forms may be sufficient.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

You start with a table that contains all the information that you want to retrieve from the database. To continue the college-database example from earlier in this chapter, a student's report card might contain several pieces of information: age, name, a list of classes, class names, teacher names, and so on. This table probably contains duplicate elements, such as class names, class identifiers, and teacher names. Therefore, you need to break certain fields into separate tables via normalization.

The process of gathering data into a single logical table is called *denormalizing,* and denormalizing improves query performance. NoSQL databases that are designed to support queries from the start often denormalize the data based on the anticipated queries to be made against that database. The student/class example illustrates the issue: Should you make `Classes` a separate document collection, or should you embed the `Class` information in the `Student` document?

If you design a database by thinking about the queries instead of thinking about the data, you may decide that because most of the application queries concern students, embedding the class information in the student document makes sense. You must also consider whether the student is enrolled in a large number of classes (and *large* is determined in part by any restrictions placed on the size of a particular document). On the other hand, if you expect many queries that concern only classes, you may create `Class` as a separate collection.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 3

Going Beyond the Data Center with NoSQL

In This Chapter

- Seeing how NoSQL distributes data
- Making sure that data is consistent
- Keeping document contents durable

Even though data centers are familiar features of enterprise computing, they've always had some drawbacks, especially in the early days. To support large databases, NoSQL runs on a large number of networked servers organized as clusters (even if the servers are mounted in the same rack in the same server room). Now, granted, some problems can arise when you distribute data across multiple machines. For both a relational database management system (RDBMS) and a NoSQL database management system (DBMS), maintaining availability and consistency across multiple servers continues to be a challenge. This chapter looks at how NoSQL meets that challenge.

Data Distribution

Data is primarily distributed in two ways:

- *Sharding* — different data on different nodes
- *Replication* — copying the data to multiple nodes; replication can be either peer-to-peer or master/slave.

With NoSQL, you can use sharding or replication, or both. Different NoSQL databases provide different functionality. This section describes some common approaches.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Sharding

Sharding (also called *horizontal partitioning)* involves partitioning the database on the value of some field. Various documents in the database are stored on separate machines based on the value of that field, but sometimes, an application that accesses the database makes that decision; other implementations place an intermediary between the application and the databases running on the remote nodes. Some NoSQL databases require the admin to configure sharding rule; others will automatically rebalance data across nodes proactively. The downside of manually configuring rules is that you may inadvertently have an imbalance of data between nodes. This could result in a slower response time from some nodes. An RDBMS supports partitioning, but usually, the partitions remain on a single machine (in the form of particular disk partitions).

You can use various techniques to partition your NoSQL database:

- **Use a key value to shard your data.** In this approach, for example, all customers who have their home offices in California are stored on a node in your California data center.

  One way to store data in this way is to use range partitioning. In *range partitioning,* specific ranges of data are stored on special servers that you can consult to see which servers are responsible for which ranges.

- **Use load balancing to shard your data.** In this scenario, a database may split a large shard into a series of smaller shards that run on multiple machines. This process may not be visible to the application developer because the DBMS makes the decision regarding performance.

- **Hash the key.** In this scenario, the key value is mathematically transformed to (ideally) a shorter value. *Consistent hashing* assigns documents with a particular key value to one of the servers in a *hash ring* (a collection of servers, each of which is responsible for a particular range of hash values). Again, the details are hidden from the application programmer.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Replication

NoSQL uses two forms of replication: peer-to-peer and master/slave. Both methods have problems with consistency, but for different reasons. That's why, if you can implement a solution on a single machine, you may want to do exactly that and use more traditional means (such as backups) for database recovery.

Certain NoSQL databases, especially graph databases (see Chapter 1), are quite happy to live on a single server. They're still vertically scalable but can perform queries only on a single node. A change is committed to a single node, and that change is replicated to other nodes. Because most NoSQL databases are intended to run in a cluster, you may face replication issues sooner or later. Usually, this is eventually consistent (see the discussion of ACID and BASE in Chapter 1), so some nodes lag in passing that data back in query results.

Eventually consistent means the state of the database isn't consistent until the change is committed to the replica node.

Master/slave replication

Replication is simpler in a master/slave configuration: All modifications are made on the master node, and these changes are pushed out to the slave nodes. A problem occurs if the master fails before the slaves are modified; then the slaves elect a new master, and updates continue. When the old master is brought back, conflicting changes may need to be reconciled, especially if multiple nodes are updating the same partition. In the best-case scenario, the master handles all writes, and reads can be shared across the slave servers. With that said, master/slave reconciliation isn't needed if the NoSQL server promotes a replica to the new master and is ACID compliant — and then no inconsistencies are possible.

Peer-to-peer replication

Peer-to-peer replication means that no master node exists: Any node can accept reads or writes. Writes are propagated to each peer, and reads can occur from any of the servers.

Replication is the workhorse of many NoSQL databases. But the main problem in replicating the database on multiple nodes is that these replicas can become inconsistent. Consistency becomes an issue when two users update the same document

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

at the same time on different peers. A read inconsistency can occur if an update to a documented hasn't propagated from one peer to another: One person sees one value returned, and another person sees a different value. Trouble? That depends.

An example of inconsistency could be comment counts on blog posts. The count could be different for different people. With low-value data, that may not be a problem. However, an inconsistent view of inventory may cause a problem. For example, if one person sees one count and the other is actually lower, you may be surprised when you run out of inventory.

The RDBMS solved this problem a long time ago by using journals and log shipping among clusters. Some NoSQL solutions are starting to provide this feature, but you may have to consider handling the problem in application development.

The ACID Test: Data Consistency

ACID is a way to structure a database to keep transactions reliable. The acronym is short for *atomic, consistent, isolated,* and *durable,* as follows:

- *Atomicity* means that the effect of the transaction is *atomic* — that is, all or nothing. If one part of the transaction fails, all parts fail, and the state of the database is unchanged.
- *Consistency* means that a transaction causes the database to transition from one valid consistent state to another consistent state.
- *Isolation* means the end user sees results as though each transaction operated in isolation, even though many transactions may be operating in parallel.
- *Durability* means that when a transaction succeeds, the database guarantees that the results of that transaction appear in the database regardless of any other system malfunction.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

ACID constraints in NoSQL

Most NoSQL databases relax ACID constraints. A NoSQL database may guarantee that an update of an individual document is atomic — that is, it happens or it doesn't. No such guarantees can be made, however, if the logical transaction requires updates to several collections (tables) within the same database, not even if the database is running on the computer that holds the data.

When you work in a distributed computing environment, you have to balance your need for consistency, availability, and durability. To provide high availability, some, but not all, databases will relax constraints on consistency and durability.

In practice, partition tolerance is required (and even unavoidable) in most NoSQL databases because a database has to function in some capacity as much as possible. That leaves two properties to be compromised: availability of data and consistency of replicas.

The RDBMS wasn't ACID when it was originally created. It added ACID support over time. NoSQL databases are also following this trend. MarkLogic and FoundationDB provide ACID consistency.

Read and update consistency

Consistency, for a NoSQL DBMS, means update and read consistency. In the RDBMS ACID approach to database design, the goal is to make sure the data in the database remains consistent internally. In a college database, for example, when a student transfers from one section to another, consistency means that you see the student in one section or another, but never both. Otherwise, the student isn't enrolled in the class.

Here's some further explanation:

- *Read consistency* means that two readers see the same data after an update. An RDBMS achieves this task via transactions. A NoSQL database using replication, however (see "Replication" earlier in this chapter), can have a delay — the *inconsistency window* — before all replicas are updated. This approach is called *replication-consistent*.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

- *Update consistency* means that when two modifications are issued simultaneously, one update succeeds, and the other is notified that the record it's trying to update has been modified since the last read. This approach is called *update-consistent.*
- The classic case of update consistency looks like this: Charlie checks out the file `foo.c` to fix a bug. Gordon checks out the same file to fix another bug. Charlie submits his changes first; then Gordon submits his changes. Who wins? Gordon should be notified that changes were made in the original file after he checked it out for modification.

Document Durability

Durability represents whether a modified document is stored permanently when an update is finished. In a NoSQL database, durability usually is implemented via replication (discussed earlier in this chapter). Eventually, all nodes in the cluster need to have identical copies of the data, because a read operation could be served by any of the replicas and needs to get the same data from all of them.

The number of nodes required to ensure a consistent read is called the *replication factor.* Some nodes in a cluster are participating in replication. Of those nodes, you need some number of nodes to respond to an update (a write request) and some number to respond to a query (a read request). The replication factor is the number of nodes that need to participate to guarantee consistency.

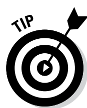
In master/slave replication, you need to have only one node — the master — respond to an update, because you trust that the changes will be sent to the slave nodes.

If you have multiple readers (R) and writers (W) participating in replication, you need to have more than half the total number of nodes (N) respond to the update to guarantee that the data has been saved. This situation is called a *write quorum* ($W > N/2$). A database designer can play with these numbers to achieve different effects, such as read consistent, always consistent, or eventually consistent.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Some math here: The strongest consistency (N = 3, W = 3, R = 1) guarantee occurs when the number of writers equals the number of nodes performing replication. If N = W, every server needs to confirm the write before the write is acknowledged back to the application. By definition, therefore, all machines are in sync. Eventual consistency occurs if W = 2 and R = 1 and N = 3 (a write quorum). When the third node rejoins the set, it 's updated based on the data written to the other two nodes.

In a traditional RDBMS installation, durability is achieved by having the underlying database engine ensure that the write has succeed. Another technique ensures that the transaction has been saved to a *journal file*. This file contains a list of transactions that have been applied to the database since the last update was written to disk. If the database crashes before the update is written to disk, the journal file can be replayed against the database when it restarts to bring the database back into sync.



Some NoSQL DBMSes, such as MarkLogic's, offer journaling capability. If you're running mission-critical applications and need a guarantee that each transaction is committed, take a look at ACID-compliant NoSQL databases.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 4

Seeing What Enterprise NoSQL Can Do

In This Chapter

- Defining enterprise NoSQL
- Using replication
- Backing up data
- Ensuring data integrity
- Spotlighting critical security matters

Without question, NoSQL databases have proved their worth in enterprise computing. The earliest examples of databases that came to identify the NoSQL movement — Google's BigTable and Amazon's Dynamo — were used by the world's largest search enterprise and the world's largest online retailer.

Using NoSQL for internal data management for business intelligence, however, is different from using it for customer-facing applications. If you plan to use a NoSQL database management system (DBMS) for mission-critical projects, you need to consider the costs across the entire useful life of those projects.

In this chapter, I discuss *enterprise NoSQL* — a type of NoSQL DBMS that provides enterprise-class performance, security, and reliability. Enterprise NoSQL should be the kind of DBMS on which you'd be comfortable betting your business. For that reason, it needs to offer the capabilities listed in this chapter.

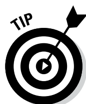
These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Enterprise Replication

Replication (see Chapter 3) helps many an enterprise achieve its goals, including high availability, robust disaster recovery, and fast performance. Replication makes demands on an enterprise's data and communications infrastructure, however.

At its base, replication occurs when a master database — the database being replicated — creates a copy of a new or updated document in a replica database.

Many NoSQL databases also provide something called *local disk replication.* This replicates data within the same cluster to different nodes. Depending on the NoSQL database, some use this to provide extra read replicas, and others do this in case one motherboard dies. In this case, a secondary node takes over as the master for the replicated set of data. When the first node is fixed, it can take a fresh copy and become the master for this data set again. This feature isn't required when nodes use centralized NAS storage rather than local attached disk storage.



Some NoSQL databases like MongoDB require dedicated read replica nodes that are only queried if a primary node fails due to hardware failure. This means you're paying for redundancy only. Other NoSQL databases, like MarkLogic, store secondary replicas that coexist on the same physical host as a primary host for a different part of the same database. This reduces the overall number of servers required while still providing local disk failover.

Replication strategies between NoSQL databases vary. Some don't guarantee the order of the updates (meaning two updates to the same document could arrive at the replica in a different order) or are transaction-aware (all updates that are part of that transaction are applied as a single transaction on the replica).

Replication can also be synchronous or asynchronous. With synchronous the primary site doesn't complete a transaction until the secondary site is updated. This can lead to slower update response times. Many NoSQL database are instead asynchronous. Asynchronous will finish a local transaction and then replay the journal frames at some point afterwards on the remote replica cluster.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Example enterprise-class replication

MarkLogic offers an example of enterprise NoSQL and shows what you need to consider when setting up support for enterprise-class replication. In this DBMS, replication is implemented by copying journal frames and replaying them to a replica database. Replica databases can be queried but not updated by the application. If any content is stored in the master before replication is enabled, that content is bulk-replicated in the replica database. Bulk replication also occurs when the master and the replica have been detached for so long that journal replay isn't possible.

It's possible to be asynchronous and transaction-aware — the NoSQL database, like MarkLogic, guarantees the updates are applied in order. This ensures the remote replica is still consistent with the primary (albeit slightly delayed). Asynchronous replication also allows the database and replica to still operate when network latency is high.

Replication supports business continuity and disaster recovery by enabling you to keep a replica of the data outside your primary data center. By outside, I mean at least hurricane width away. Usually, 40 to 100 miles is considered to be sufficient distance from areas where natural disasters are possible.

Enterprise Data Backups

An enterprise needs consistent, database-level backups, including data, schema definitions, configuration details, and journals, and so on. Though you can back up only some of the data or back up portions of it at different times, you still need to be able to back up the entire database in one fell swoop.

NoSQL databases can offer backup capabilities via database-supplied tools or file-system-level backups (such as LVM in the Linux world). NoSQL systems that can synchronize configurable database journal archiving can provide point-in-time recovery of specific data from complete backups or from snapshots. Being able to specify the lag time between journal

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

data being written to the active log file and data being sent to the backup journal allows you to manage your recovery objectives.

Multiple-Record ACID Transactions

Some NoSQL databases offer atomic transactions within a particular document or record, meaning that an update on a single record will happen or it won't. Keep in mind, however, that the same guarantee doesn't apply when multiple documents are involved in the update.

Consider the case of a payroll update. You're updating the raise percentage for employees based on their performance reviews, as illustrated by the following SQL pseudocode:

```
Update raise_percent=0.20 in employee where
performance_review = 'Superior'
```

In this case, you want to ensure that every employee who has a superior performance review gets a 20 percent raise. Ideally, you want the update to happen fast, which is a problem if the system is looping through every document and updating the `raise_percent` attribute. If the check-writing program is processing the same collection at the same time, Jane Q. Developer may not get the raise she deserves, and that's definitely not what you want.

Here's another example. Suppose that you have two collections in your college database: one for students and one for courses. You made this design decision because your applications need to access course data separately from student data. Now suppose that you want to add a new course section and reassign selected students to this new section. You want to make this change atomically, creating the new section and moving students into it, or failing to add the new section and leaving the students where they are. You don't want to have only some students reassigned, and you don't want to have any student dropped entirely (meaning that he or she no longer has an entry corresponding to that course).

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

A NoSQL database that supports multiple-document ACID transactions (see Chapter 3) won't have any trouble with either of these cases. A NoSQL database that implements atomic updates only for single documents, however, requires extra application support.

Enterprise-Class Security

Security is often one of the last considerations when it should be one of the first. Within the database world, identity and access management (IdAM) systems provide a critical service in support of security. Government-grade security ensures authentication and authorization down to document level.

Authorization and accounting

Authorization and *accounting* (or *auditing,* or *accountability*) have to be combined to determine which person or machine is allowed to do what to which data. In a typical relational database management system (RDBMS), permissions can be applied at database, table, record, and field level; allowed actions can be create, read, update, or delete. More often than not, though, permissions management is delegated to the application.

Managing these permissions is a way of enforcing the organization's security policy. MarkLogic's IdAM system acts as an enforcement mechanism for that written policy.

Make sure that you know when someone is talking about a security policy, as opposed to a set of rules that can be applied to the database to enforce a particular behavior or to notify operations of a condition that needs attention.

Authentication

Although some systems allow database access based on machine login systems, others require the DBMS to reauthenticate users before granting them access. Likewise, access rights to the database are determined by mechanisms inside the database, not by permissions granted by the machine's operating system. This is as it should be: You don't want an unauthorized person who has gained access to a particular machine to have access to the database as well.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

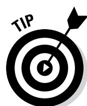
MarkLogic's system implements the Kerberos authentication protocol, which provides mutual authentication between a user and a particular server, as well as server-to-server authentication and certificate-based security. This kind of authentication can be required in addition to machine-level login authentication, which provides another level of security.

Access control

When your system has validated a particular user's digital identity (keeping in mind that a user can have multiple digital identities), the next question is to ask what privileges are associated with that identity.

In the Linux world (Linux is a common mechanism for managing permissions on files/documents), permissions are associated with particular file-system objects. Permissions for files are divided into three classes: `owner`, `group`, and `world`. Each class can have permission to read, write, or execute a file-system object. (*Execute* means something different when it's applied to a file-system directory.) Permissions are determined by aggregating the permissions for `owner`, `group`, and `world`, and making changes in permission sets is complicated by having to list all permissions for everyone.

Luckily, *role-based access control* (RBAC) can come to the rescue. With RBAC, you can set database permissions based on user roles rather than specify permissions on a per-user basis — a task that's harder to manage. One major benefit of RBAC is that a change in permissions for a role affects all users assigned to that role, which ensures that all users in that role have the same permissions (and also illustrates the "One fact in one place" principle).



Supporting RBAC in the NoSQL database reduces the amount of code in the application. This is a benefit of MarkLogic as an Enterprise NoSQL database.

Compartment security

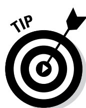
Compartment security is a technique whereby a document is assigned to one or more compartments, and the compartment is represented by a label. Roles are extended to include

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

a compartment designation as well. To access a document, a role must have the appropriate access to the document and must be in all the appropriate compartments as well. This system varies from traditional RBAC systems, in that it applies `AND` logic to the roles assigned to compartments, whereas normally, `OR` logic is applied.

Consider a `Classification` security compartment, containing the roles `unclassified`, `confidential`, `secret`, and `top-secret`. You may also have another compartment called `Department`, containing roles called `Finance`, `HR`, and `Engineering`. You have a design document for a new product that will make your company millions, and you assign this document to both the `top-secret` role and the `Engineering` role. `HR` roles that have `top-secret` access don't have access to this design, and `secret` members of `Engineering` don't see it either. Only those who have both `top-secret` and `Engineering` roles can access the document.

Compartment security can be applied to many corporate and not-for-profit information management structures. In the United Kingdom, for example, doctors only have access to certain types of documents like scan and test results but not profile information docs. So when they're trying to diagnose a patient they need to see only tests and scan results for a particular condition; they don't need to see information about income or any psychological evaluations.



To ensure security of information in a complex environment, you should consider Compartment Security in your application and an enterprise NoSQL database.

Policy Management

Policy management supports automated actions based on specified policies. A policy may indicate that only users in a particular role can update a particular database, for example, and a security warning should be generated if anyone else attempts this type of access.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Scripting use cases

Providing scripting capabilities for administrative functions eases the burden on database administrators because they can create a single script that can run across multiple machines. The MarkLogic features in this sidebar may exist in other NoSQLs, but they currently all exist in MarkLogic.

MarkLogic admin APIs provide these features:

- Configuring multiple identical instances of MarkLogic across a cluster or multiple clusters to maintain consistency among environments
- Automating server maintenance, such as backups based on criteria other than a schedule
- Managing server resources
- Making bulk updates in server configuration, such as changing roles across a large subgroup of users
- Generating log and/or email notifications for specific server events

Tying scripting capabilities to specific server events provides enhanced capabilities for automatic reprovisioning. A database administrator can use MarkLogic scripting support to grow or shrink a cluster on demand or based on changes in CPU use over time.

Administration and Management Tools

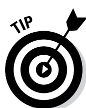
An enterprise-class NoSQL DBMS should have a suite of database administration tools, as well as other management and monitoring tools. These tools allow businesses to do the following things (among others):

- Determine a database's health at any time
- Determine proactively whether changes are required to improve performance
- Initiate any needed configuration change based on database load and resource consumption

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Sometimes, you have to make the tools yourself (or your IT department does) to fit the needs of your business. Management and monitoring tools that are based on an application programming interface (API) offer a distinct advantage to large organizations that already have a large installed base of management and monitoring systems. Using an offered API, an organization can quickly integrate information about the operational state of the NoSQL DBMS into its existing workflow and management consoles.

An enterprise-class product helps an organization create its own policies — or modify existing policies — to respond to changing security and operating needs.



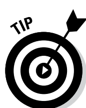
If the API results in prebuilt interfaces that connect to and use system management tools that are popular in your industry, so much the better.

High Availability

High availability is a must for business-critical applications, which must continue to function even when part of a system is lost. You must ensure that users can access data whether or not the database is on and whether or not the database has disaster-recovery support.

You can describe availability as a function of ACID compliance (see Chapter 3), high availability, and disaster-recovery replication. Journal log shipping between disaster-recovery sites and journaling on a single node ensures durability of transactions. Shared-disk systems mitigate the need for replication because shared disks are handled by the storage subsystem transparently to the database, which means that shared disks provide high availability capability.

ACID compliance, replication for disaster recovery, and high availability combine to meet the availability requirements needed to run a database that needs to run 24/7. And that's what enterprise NoSQL is all about.



MarkLogic, for example, provides multiple features that enable high availability, including fast automatic restart, automatic concurrent recovery, online backup operations, hot configuration changes, and shared-nothing architecture.

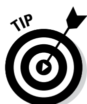
These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Scalability

You must be able to scale a database by adding or subtracting resources depending on business needs. You obtain scalability by adding more machines and replicating or sharding data (see Chapter 3). You get availability by choosing several machines to participate in a replica set. In a replica set, at least one machine should be available to respond to read requests for every part of the database (therefore avoiding partitions on read).

Integration with Third-Party Solutions

Integration with third-party services supports the integration of your NoSQL DBMS with existing management and monitoring solution. Other systems include Hadoop for batch processing and executing document enrichment functions asynchronously. You may also wish to integrate your existing relational business intelligence tools to your new NoSQL solution. Some support ODBC and native connectors to tools like Tableau for this purpose.



MarkLogic integrates with Nagios and HP Openview monitoring solutions and integrates with 3rd-party tools that have a REST interface.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 5

Real-World Enterprise NoSQL

In This Chapter

- Consolidating data
- Increasing situational awareness
- Securely storing operational data
- Repurposing data for multiple uses and applications

This chapter helps you understand what NoSQL offers you. I present use cases in which a NoSQL solution can provide four capabilities for your business: data consolidation, situational awareness, storage of operational data, and discovery and reuse of existing data.

Consolidating Disparate Data

Data consolidation means bringing together different data sources to get new value or meaning from them. These different types of data — such as photos, document contents, sensor data, and posts to social media sites — can't be stored in a relational database management system (RDBMS). The ability to search multiple data sources in one database, however, significantly improves search results and analytics.

County government

Fairfax County is one of the largest counties in the U.S. with more than 1 million residents and a budget larger than that of four states. It's also the largest jurisdiction in Virginia and

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

the Washington, D.C., metropolitan area, with a population exceeding that of seven states. The median household income of Fairfax County is one of the highest in the nation, and more than half its adult residents have four-year college degrees or secondary education. Fairfax is a diverse, thriving county, home to the headquarters of U.S. government agencies such as the Central Intelligence Agency, the National Geospatial-Intelligence Agency, the National Reconnaissance Office, the National Counterterrorism Center, and the Office of the Director of National Intelligence.

Problem

Fairfax County was challenged to respond to the high volume of requests from its constituents — government officials, community members, and developers — for property history information. Volumes of data were stored in disparate databases, file systems, and formats, so that data couldn't be searched effectively (see Figure 5-1).

Figure 5-1: Traditional RDBMSes need a lot of ingest time, data manipulation, and schema development.

Figure 5-1 shows what companies often have today: several different silos of (typically) relational data. But relational data usually amounts to only 20 percent of an organization's total data. In addition, an RDBMS can't store Microsoft PowerPoint files, the contents of PDF forms, documents, or images. This form of data management is both ineffective and costly. Like many companies, Fairfax County needed to consolidate data in one place.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Solution

Fairfax County needed a solution that would integrate disparate data, maintain its integrity, and make it easy to access. It also needed a solution that was a good fit its their existing IT infrastructure, as well as a low and predictable total cost of ownership.

The county ultimately selected a MarkLogic solution, which is schema-agnostic, accepting data from all the different silos in its current form and providing immediate access to that data (see Figure 5-2).

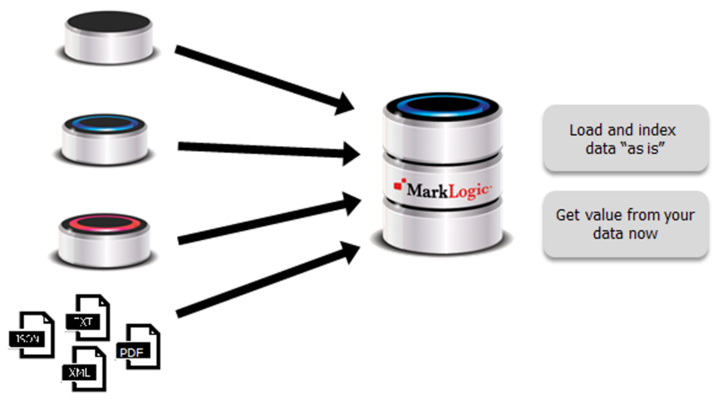
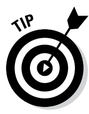


Figure 5-2: Consolidating disparate data in a single DBMS.

Fairfax County developed a repository of land-use data that made it easier for county employees, land developers, and residents to access real-time information about zoning changes, county land ordinances, and property history. This repository, held in the new NoSQL database, boosted the productivity of county employees, who no longer had to search multiple databases for pertinent information. It also enabled county residents to access information on their own computers and smartphones.



Benefits

Within six months, Fairfax was able to do the following:

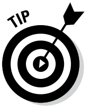
- Maintain and even increase service delivery to constituents, supporting open-government goals and enhancing its reputation with citizens and businesses
- Reduce overall cost of ownership and IT resources
- Quickly develop new applications using the same data

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Investment banking

A Tier 1 investment bank used 20 different RDBMSes as operational data stores for over-the-counter derivatives processing. Each database had its own method of ingesting the trades, and development time was required to translate each new type of derivative to the relational structure. In many cases, trades were simply shoehorned into existing schemas. As a result, a single column may store several types of information based on other parameters of the transaction, which made reporting complex and subject to errors.

The bank successfully consolidated all 20 operational data stores into a single MarkLogic database management system (DBMS) that saves all trades in their native forms without requiring extensive development cycles. Because the new product supports ACID transactions and replication, the bank maintains the reliability and availability guarantees of its DBMSes.



The new DBMS also benefits the bank in several other ways:

- **Faster time to market:** New instrument types are supported in a matter of hours, rather than days, weeks, or months.
- **Higher data quality:** A unified operational trade data store eliminates processing errors associated with multiple stores containing conflicting data.
- **Lower cost per trade:** Hard and soft costs associated with maintaining multiple systems (and nine full-time support positions) were eliminated, along with expensive exception-management costs.
- **Greater agility:** The bank can meet changing business needs such as escalating regulatory pressures without adding expensive resources.

Sharpening Situational Awareness

Situational awareness implies responding in real time to new data in the system *(alerting)* or modern streaming data, such as social media posts and sensor data. This data is passed immediately to other systems (such as military commands or

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

trading systems) or to other devices (such as battlefield-soldier devices, instant-messaging clients, or email).

The Federal Aviation Administration (FAA) brings together a wide variety of fast-changing information from internal systems and public information sources to build a real-time crisis-management dashboard. The FAA uses the Emergency Operations Network (EON) real-time application for monitoring and tracking severe weather and emergencies to keep passengers, flights, and airports safe.

To properly manage emergencies on the ground and in the air, the FAA required EON to handle multiple document types and feeds, and to make sense of the information as quickly as possible. It needed a solution that provided a back-end repository and searched for unstructured information: tweets, weather reports, news stories, emails, Microsoft SharePoint documents, geospatial data, and more. All these data sources had to be pulled together and then integrated with custom and commercial applications so officials could analyze and share the data rapidly.

Within a few weeks of implementing a MarkLogic NoSQL DBMS, the FAA had access to full text, metadata, faceted search, alerts, and complex content, and was able to roll out a fully integrated, web-based emergency communication system based on the new DBMS. The EON dashboard now provides a single consolidated view of situational-awareness data. The new database enables the FAA to load, search, transform, and render multiple types of unstructured data quickly and easily to provide a consolidated view of situational-awareness content (see Figure 5-3).

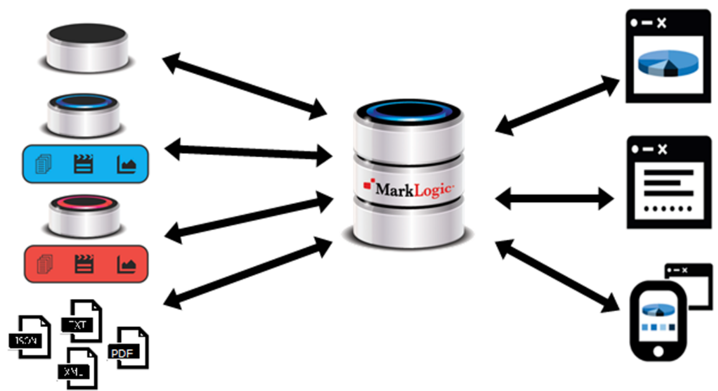


Figure 5-3: Sometimes, it just makes sense to store disparate data in one place.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Storing Operational Data

An *operational database* is a highly scalable yet enterprise-ready database for rapidly added, high-volume information that scales horizontally on commodity hardware. Examples of operational databases include trading platforms, social media, and analytics platforms.

One large financial institution carries out hundreds of thousands of trades every year, with many complex trades taking an average six months to close. In these complex deals, each action is represented as a document in FpML (Financial Products Markup Language) format — an open-source, XML-based trade description standard.

Unfortunately, this approach created several problems for the organization:

- Many institutions with which this organization interacts use different versions of the FpML standard.
- The FpML standard itself changes over time.
- Each trade document contained only a trade instruction — not the full background information required for a person to carry out the trade or determine its risk. This so-called *reference data* on companies, exchange rates, and other vital information was held within an RDBMS.
- Trades weren't always visible and actionable as soon as they were committed to the database, which created potential risk of losing documents after they were committed.
- The regulator requested a 5-minute view of the risk to which the institution was exposed, not the 24-hour view provided by its traditional data warehousing approach.
- The organization itself wanted to be able to spot risky trade behavior in time to stop any risky transactions.

A MarkLogic enterprise NoSQL DBMS was a good fit for solving these business problems:

- It natively understands and can index entire XML documents.
- Changes to the FpML schema over time don't present a problem. The DBMS can use field, element,

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

element-attribute, and path range indexes to group common elements in different schemas.

- The financial institution can use XQuery with database triggers to perform entity extraction and enrichment, which automates the process of fetching and embedding relational reference data within the trade document.
- The DBMS is ACID-compliant and highly available, and it ships logs to a disaster-recovery site, so the financial institution's trade documents are never lost after they're committed to the database. (For details on ACID, see Chapter 3.)
- Using the same range indexes, the DBMS can provide immediate, live reporting on all trade data, satisfying any regulator.
- The DBMS provides built-in alerting, enabling the organization to respond quickly to any trade or activity that may be invalid or risky.

Discovering and Repurposing Data

Repurposing existing data for other uses can be a major source of new revenue for an organization. Often, a single app connects to a large number of data sources, which amounts to a lot of custom work.

Then along comes another product, followed by another, and the business ends up with a rat's nest of connections between applications and data sources (see Figure 5-4).

The problem gets worse every time a new data source or application is added. By this point, I think you can guess what a headache the solution is to manage.

A MarkLogic solution allows businesses to solve this problem without the headache. Compare Figure 5-5 with Figure 5-4. The solution in Figure 5-5 loads data from many sources without requiring conversion and provides that data to many device types and form factors in real time, on the scale that large organizations require.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

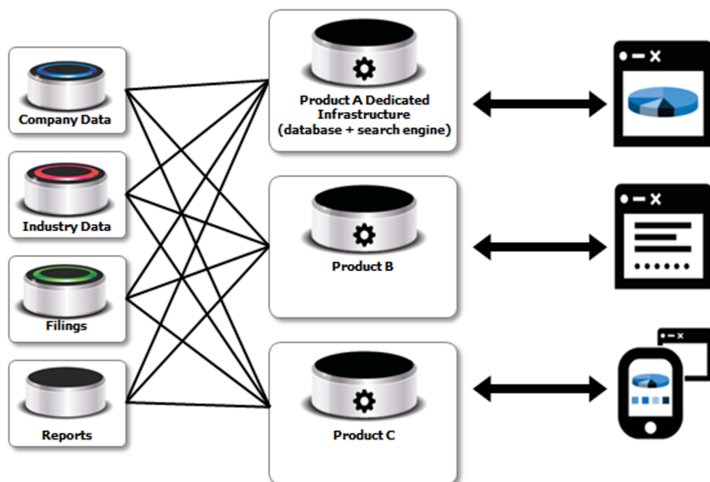


Figure 5-4: Complex connections can be expensive to untangle.

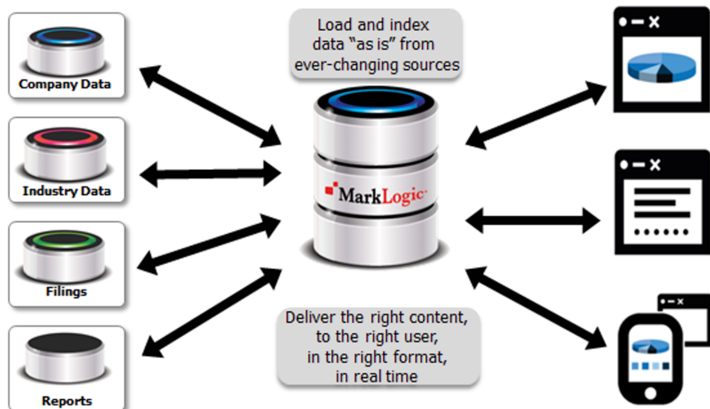


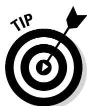
Figure 5-5: Streamlining connections also simplifies data delivery.

In this section, you take a look at a couple of examples of companies that put MarkLogic solutions in place and came out feeling less scrambled.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Publishing

Springer is a leading publisher of scientific, technical, and medical books and journals. Its customers were begging for a web-based solution that was part of their workflow and also provided a single piece of content. The company had built a database of 3 million images, along with associated XML descriptions, but integration of this content into existing workflow was difficult.



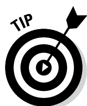
Springer migrated its existing system to MarkLogic and realized the following benefits:

- Increased user time spent on the website
- A 50 percent decrease in bounces
- Storage space in the new database for more than 5 million images

Broadcasting

The British Broadcasting Corporation (BBC) regularly repurposes content. Editors put content into a database, and that content gets published dynamically. (Dynamic publishing is content repurposing on steroids.)

During the 2012 Summer Olympics in London, the BBC used a MarkLogic NoSQL DBMS to publish content dynamically, based on user settings and preferences. The solution also incorporated photographs, results, and details about the athletes alongside editors' content to provide dynamically repurposed content. You can see how this solution works on the BBC Sport website today at `www.bbc.com/sport`.



Discover more information at `www.marklogic.com/resources/playing-to-audiences-in-the-digital-world`.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Chapter 6

Ten Questions to Ask About NoSQL Solutions

In This Chapter

- Considering your internal and external needs
- Calculating hardware and software requirements
- Looking forward to growth and innovation

When you're considering NoSQL solutions for your application or system architecture, the choice is rarely simple or certain. NoSQL databases provide certain benefits, but they don't offer everything — and they don't try to.

This chapter examines ten factors you need to weigh when you're considering NoSQL products. Working your way through this chapter takes you a long way toward finding the right solution.

Are You Ready to Incorporate the Solution?

Incorporating a NoSQL database management system (DBMS) into your business is disruptive. The technology itself isn't disruptive; the changes that you're bringing in with it are. With NoSQL, you're seeking to do new things with data, and anything new involves a learning curve. The only question is how steep that curve will be. You (or your IT staff) have to be able to do the following things:

- Modify your application design procedures and your system application architecture to accommodate the capabilities and operational characteristics of the new software.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

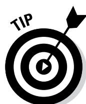
- Understand the NoSQL DBMS, as well as the languages needed to query the database. In some cases, you may have to use Java to write to the NoSQL application programming interfaces (APIs); in other cases, you may have to know a completely different query language. (Most NoSQL DBMSes, however, provide a cheat sheet from SQL to the native query language.)

If you don't have these skills, you need to have a plan to develop them internally or acquire them. In either case, allocate training time and budget into your plan.

Is the Solution Agile Enough?

To put this question a little differently, can the solution support documents *and* values *and* triples *and* search, *and* is it schema-less, *and* does it provide multiple views of all this data, with the same features implemented consistently? Buying a NoSQL database that can manage many different data types can be better than buying four — one to manage each data type — and then integrating them to a single application. The answer can help you determine exactly how disruptive bringing a NoSQL solution into your organization will be (see "Are You Ready to Implement the Solution?" earlier in this chapter) and whether the solution can expand with you as you discover new ways to mine your data.

Incorporating a NoSQL solution into your business requires willingness to change, adapt to change, learn new skills, and think about solutions to business problems in a new way. You must decide whether a NoSQL solution is worth the effort in terms of the cost of disruption and whether you'll ultimately achieve a significant return on this investment.

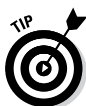


Ask your vendor how quickly its customers have been able to implement their first projects, as well as how much development and training time was required, and whether consultants were used. Also find out how quickly customers were able to implement their next projects and whether they needed the same level of support.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Does the Solution Meet Your Application Requirements?

You may be looking at a NoSQL DBMS to solve an existing business problem, or you may be anticipating a change in the number of users or the type and amount of data that you need to support. Perhaps you're running a pilot project to gauge the time and effort required to develop a prototype, to test performance in your anticipated operational environment, or to build an environment to evaluate different NoSQL solutions.



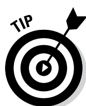
Whatever your reason for considering NoSQL, it pays to know what problem you're trying to solve and what you expect in terms of costs (both development and operational) and performance.

How Much Development Is Required to Get Data In?

Some NoSQL databases require you to ensure that data is in a particular format and some only require the files to be a specific type. Depending on how the database works, the speed with which the data is available to be searched and can connect to an application will vary.

Ask yourself some questions here:

- Will the database immediately fit your application needs, or will it need to be customized?
- How much, if any, effort will it take to develop the Extract-Transform-Load (ETL) software?
- Does the vendor supply that software or training?
- Do you have the expertise in-house, or will you have to hire consultants?

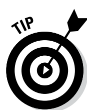


Ask the vendor to show you how that data can be brought into the database with ease and how quickly the data can be accessed after it's in the database.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Do You Need New Tools?

Integrating NoSQL into your development and operations organizations may make certain tools obsolete, so you may have to acquire different tools. You probably can extend your development environment to support the language in which you need to write your application code, however. Eclipse, for example, has modules for multiple languages, as do Komodo Edit and Visual Studio, and all can be extended via external libraries.



Working through how you'll integrate a new DBMS into your existing management and monitoring systems helps you identify the obstacles you may face. In this situation, vendor-offered interfaces to popular management and monitoring systems can be life-savers.

Does the Solution Support Your Cloud Strategy?

Your organization does have a cloud strategy, right? Chances are that it does, even if that policy isn't written down or even articulated. Figuring out whether a potential NoSQL solution supports your cloud strategy requires you to know what kind of cloud platform you're considering: IaaS (Infrastructure as a Service), PaaS (Platform as a Service), or SaaS (Software as a Service). You also need to know how you plan to use the cloud with your NoSQL DBMS, such as replicating and backing up data.

If you're using a public cloud solution, you also have to address security concerns regarding data in transit and data at rest — concerns that you may not have had to consider while operating inside the bounds of your data center or using a private cloud solution. Both you and your vendor need to address encryption, performance, and access-control concerns.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Is the Solution Enterprise-Ready?

Enterprise NoSQL characteristics (see Chapter 4) include backups, flexible replication, multiple-document ACID transactions, and security.

When you're considering an enterprise NoSQL solution, also consider the following:

- What kind of track record the vendor has (including how long it's been doing business)
- Whether the vendor's existing clients are similar to you
- How well the DBMS fits with your product-development cycles and ongoing maintenance needs
- Whether the vendor offers help-desk support and problem resolution directly or through a third party and are they based in your country
- Whether the DBMS can be customized and, if so, who can do the work
- Whether the vendor provides DBA tools to manage the health and deployment of a database cluster

Do You Need Specialized Hardware?

One advantage of most NoSQL solutions is that the DBMS runs on commodity hardware and doesn't require specialized hardware. Commodity hardware isn't necessarily cheap, but the good news is that you can expand as your needs grow, you don't have to pay for more resources than you're consuming, and you can repurpose hardware if your consumption drops.

If a solution provider wants you to buy specialized hardware to support the database, be very careful. This just isn't the "NoSQL way" for increasing capacity, and you should be prepared to question the vendor closely about why its solution needs specialized hardware.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Will the Solution Grow with Your Business?

If you have a Big Data problem — that is, if unstructured data that's increasing in volume, variety, velocity, and complexity is critical to your business — understanding how a NoSQL solution will handle increasing loads is extremely important. Find out the following things:

- Whether the vendor has enterprise customers who are using the solution in scale and in production for critical data
- Whether the vendor's customers are increasing its use of the DBMS in its environments
- Whether any customers are using their product on a public site that you can access and try for yourself

Can the Solution Speed Application Development?

Some NoSQL databases don't include security, consistency, and search, which can incur a lot of overhead in development cycles. Consider using a NoSQL database that provides the enterprise features you need to assist in application development, such as built-in advanced query capabilities, word stemming, full text search, geospatial radius, and alerting.

A database that provides rich application libraries and functionality can dramatically reduce implementation times, from as long as 18 months to as little as 6 to 12 months. Purchasing a commercial NoSQL database rather than an open-source alternative can pay huge dividends in the end. Try out MarkLogic's commercial enterprise NoSQL database to reduce your implementation time.

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Notes

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Notes

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Notes

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

Notes

These materials are © 2014 John Wiley & Sons, Inc. Any dissemination, distribution, or unauthorized use is strictly prohibited.

What's the question you've always wanted to ask of your data?

Most people think they have access to all the data that they can. But that's usually only true for data that conforms to the structure and data model required for a relational database. The new generation of NoSQL databases are turning that perception on its head and speeding up development time dramatically.

- ***Make the most of all of your data* — schema-agnostic NoSQL databases don't put limits on the data**
- ***Deliver applications with speed and agility* — don't be slowed down by time intensive ETL or data modeling**
- ***Solve new problems* — without the limits of a relational schema, you can deliver apps you never thought possible**
- ***Make smart choices* — decide which NoSQL database is the right one for your project**

Open the book and find:

- **An explanation of NoSQL jargon**
- **Why Enterprise NoSQL is necessary**
- **What to look out for and why**
- **What's possible with NoSQL**

Go to Dummies.com®
for videos, step-by-step examples, how-to articles, or to shop!

FOR DUMMIES®
A Wiley Brand

ISBN: 978-1-118-82900-4
Not for resale