Building Scalable, Flexible Database-Based Applications

Developing with

Couchbase Server

O'REILLY®

MC Brown

Developing with Couchbase Server

Today's highly interactive websites pose a challenge for traditional SQL databases—the ability to scale rapidly and serve loads of concurrent users. With this concise guide, you'll learn how to build web applications on top of Couchbase Server 2.0, a NoSQL database that can handle websites and social media where hundreds of thousands of users read and write large volumes of information.

Using food recipe information as examples, this book demonstrates how to take advantage of Couchbase's document-oriented database design, and how to store and query data with various CRUD operations. Discover why Couchbase is better than SQL databases with memcached tiers for managing data from the most interactive portions of your application.

- Learn about Couchbase Server's cluster-based architecture and how it differs from SQL databases
- Choose a client library for Java, .NET, Ruby, Python, PHP, or C, and connect to a cluster
- Structure data in a variety of formats, from serialized objects, a stream of raw bytes, or as JSON documents
- Learn core storage and retrieval methods, including document IDs, expiry times, and concurrent updates
- Create views with map/reduce and learn Couchbase mechanisms for querying and selection

Twitter: @oreillymedia
facebook.com/oreilly

US $19.99 CAN $20.99
ISBN: 978-1-449-33116-0

O'REILLY®
oreilly.com

Developing with Couchbase Server

MC Brown

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Developing with Couchbase Server

by MC Brown

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette

Production Editor: Christopher Hearse
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

February 2013: First Edition

Revision History for the First Edition:

2013-01-31 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449331160* for release details.

ISBN: 978-1-449-33116-0

[LSI]

Table of Contents

Preface

Introduction

As a long time developer of database-based projects, I can appreciate how over time we have been programmed to work within the table and row-based structure provided by systems like MySQL and Oracle. These database engines all work by using SQL to define the structure and schema, and then ultimately to insert, update and retrieve information.

SQL is not going to go away any time soon, and there are plenty of solutions and use cases where the SQL and the rigid schema make sense. But there also occasions when the strict structure and schema are more of a hindrance than a benefit. Nowhere is this seen more clearly than in the recent explosion of highly interactive websites and social media. The variety of information, and much higher levels of interactivity require a completely different type of database in order to handle the load. This is especially true if your application suddenly becomes hugely popular and the volume of activity explodes.

Couchbase Server 2.0 is designed to help in those situations. As one of the growning number of NoSQL-based database technologies, Couchbase Server takes a different approach to the prbolems of storing, searching and supporting data it's availability. By using intelligent clustering to distribute and share the load, Couchbase can handle the database needs of those highly interactive applications.

Developing applications against Couchbase Server requires knowing the new structure and interface. Gone are the complexities of SQL and the rigid schemas. In it's place, you have new flexibility in the information storage, and diferent ways for storing, reading, connecting, and querying the data that you have stored.

In this guide, I've combined some basic background information on Couchbase Server and how it operates behind the scenes, with the information you need to start building applications. The methods for storing information and querying that data, and how you can organize and format your data to get the best performance and operation from your Couchbase Cluster.

Where to Get Help on Couchbase Server

The information provided in this book is designed as a basic guide to developing with Couchbase Server 2.0.

For more detailed information on Couchbase Server, you can read the full manual. For more general information about Couchbase Server, visit the Couchbase website.

Detailed background information on building and developing applications using Couchbase Server can be found in the Developer Guide.

For information on the client libraries used to build applications against Couchbase Server, see this Couchbase page.

For a list of all the available documentation for Couchbase Server, see this Couchbase page.

To get involved with the Couchbase community, there are forums available on the Couchbase page, and a mailing list at this page.

To get in touch with the author, please contact me at *editors@couchbase.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

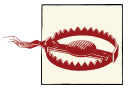
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Developing with Couchbase Server* by MC Brown (O'Reilly). Copyright 2013 Couchbase, Inc., 978-1-449-33116-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/developing-couchbase*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

The Couchbase engineering group, and the other teams in Couchbase that help define the product, support, community, and SDK development were all key in helping to produce not only the software behind the book, but also in many of the tools and examples that appear in it. It should go without saying that without their work, this book simply wouldn't exist.

Once again, the support and input from Perry Krug to help make sure the content is correct, up to date, and useful, has been immense. Without him, this book would be far less useful, not to mention inaccurate. Thanks as well to the rest of the product management team who helped to review and comment on the content.

At O'Reilly, thanks to Meghan Blanchette, my incredibly understanding and supportive editor, and Mike Loukides, who supported the inception of the book and its content.

CHAPTER 1
Couchbase Overview

Couchbase Server is a NoSQL document database for interactive applications that has a flexible data model, is easily scalable, provides consistently high performance, and is "always on", 24*7*365. From a developer perspective the key aspects are the flexible data structure and the always-on nature. The data model is flexible because at the core, Couchbase stores streams of bytes associated with a document ID. For additional features and practicality, that information can also be JSON-formatted (which brings additional functionality and benefits).

The scalability, performance and always-on nature are primarily focused on the administration and operations side, but they have one key impact on the developer experience. Couchbase supports applications and data loads with a very high level of concurrency. It is not uncommon to find applications with millions of active users, thousands (and more) of data operations per second, and working with some very large datasets.

The application *Draw Something* is a good example of a highly-concurrent application built on top of Couchbase Server. *Draw Something* was a smartphone application that allowed people to draw pictures, share it with friends, and let them guess what the picture depicted. As is often the case with such a simple premise, the game became very popular, very quickly. Within 6 weeks the game had gone from launch to an exceedingly popular application with:

- 15 million daily active users
- 3,000 new pictures generated every two seconds
- Over two billion stored drawings

To just about anybody, those are some staggering numbers. To an administrator, it becomes more staggering when you appreciate that they managed this without taking the site down, and grew from a fairly small 3 node cluster to more than 90 nodes (across

three clusters). All without taking the system offline, redeploying or architecting the core application and the database service, and still while serving those millions of picture-hungry users.

For a developer, Couchbase Server is actually a fairly straightforward application. Although the system is designed as a clustered database, as a developer this is not something you need to frequently consider. Or even, at a practical level, to be aware of. Using a Couchbase cluster with two nodes is the same as using a Couchbase cluster with 30 nodes. There are no complexities, no master/slave relationships or complex data handling that need to be supported at the client level.

To a developer, this is probably the most important aspect of the system to understand. When developing with Couchbase the most important concerns are about your application and the data that it needs to store, rather than having to design, and later cope, with sudden growth and complex and arduous scale out procedures.

Architecture and Structure

Couchbase Server is a cluster-based system where each node in the cluster is entirely equal. The software and configuration of each node is identical.

The core of Couchbase Server is a document based storage sysytem, where individual documents are identified by a unique document ID. Data is persisted to disk, and a caching layer provides fast access to the stored objects.

For a developer using a database there can be no more important connection than the one between your application and the underlying server. Couchbase Server supports a number of different client libraries that connect to the cluster. From an application perspective, the fact that the database is a cluster of individual nodes is entirely hidden from the developer. Applications read and write document objects tot he cluster, and the client library and Couchbase Server handle the distribution of information, and where a specific object is located.

Buckets and vBuckets

Two important structures internally within Couchbase Server are Buckets and vBuckets. The Bucket is a logical holder of information and is used to enable you to compartmentalize information that you store into your database. Couchbase Server supports multi-tenancy, that is, the ability for more than one application to store and retrieve information within Couchbase. Buckets are the logical elements used to support this. Buckets are:

Named
: All buckets have a name used to identify them on the system.

Shared
: All bucket data is automatically shared (and sharded) across the Couchbase Server cluster. You cannot control how the information is distributed (it is automatically controlled), but your client library will handle the communication directly between your client and each node in the cluster to identify where the data should be stored (or retrieved) from.

Optionally password controlled
: All buckets can have an optional password. This can be useful both for securely providing access to the information in your database, and to help prevent you from accidentally connecting to the wrong bucket and updating/deleting information.

Independently managed and monitored
: Each bucket has a RAM quota, replication configuration and disk storage. Each bucket can also be independently monitored, and each bucket can be independently compacted and indexed.

Applications can use one or more buckets for storing information, but it's important to remember that from a server perspective, buckets cannot talk to each other. This has particular impact on views and indexing. The current limit of the number of buckets that can be configured within a single cluster is 10.

Now we know that the information is automatically sharded and distributed around the cluster. Without wanting to get ito too much detail, the method of distribution is by dividing up the bucket content into multiple vBuckets. The vBuckets are distributed around the cluster, and because there are a fixed number of vBuckets, we can hash the document ID to a vBucket, and then through a vBucket map, map the vBucket to a node. When you store a document in the database, the document ID is processed by a hashing algorithm that produces a number, the number refers to the vBucket, and the client library uses the vBucket map to to determine which node within the cluster that is responsible for the vBuckets.

The vBucket structure is critical to the way the Couchbase Server works and is flexible and scalable. When the cluster is extended, individual vBuckets (and the data they contain) are 'rebalanced' around the cluster. Since the number of vBuckets does not change, finding the vBucket number for a given document ID has not changed. The hash is computed, the vBucket number determined, and the vBucket map is used to identify the node. Because a given document ID will always compute the same vBucket ID, and the vBucket map always describes this structure, the number of nodes in the cluster can change, shrinking or expanding to cope with different application loads.

For an analogy, let's consider the typical, physical, filing cabinet. The filing cabinet is constructed of different drawers (our nodes), and there are a fixed number of folders within the drawers (with each folder analogous to the vBucket). On the wall by the filing cabinets is a directory that tells you which drawer contains which folder. If I buy a new

filing cabinet, I can move the folders around within the drawers so that they are evenly distributed. Once I've completed that operation, I then need to update my directory (our vBucket map). To round the analogy out, there is only one copy of any piece of data within the drawers at one time. Because the data is automatically sharded and distributed, there is no problem in ensuring that the different 'versions' of the data across the database are in sync. There is only one version of the data. For additional security, however, you can use replication to create a copy of all the data in a bucket. The replicas are distributed around the stored data, just like keeping a carbon copy or photocopied version of each file in a different drawer.

As a developer, you don't need to know about the vBucket system, since your client library will handle the entire hashing/lookup and communication process for you, all you need to do is connect your application to your cluster. That said, some of the errors returned by the system will refer to a vBucket specific problem, hence why I've included this background.

Data Storage and Retrieval

The most important part of any database system is how you store and retrieve the information from the database. Couchbase Server is a document store, you store data into the database using a document ID and the corresponding document data. The document ID must be unique within the bucket, but the document value that you store can be anything—a stream of bytes, a serialized object from your application language, or a flexible document format such as JSON. We'll return to the significance of JSON in a moment.

Because of the document nature, the storage and retrieval of information is straightforward. There are no complicated queries or structures to write, and in fact, the core operations can be summarized into just four types of statements: Create, Read, Update, and Delete (CRUD). You can see these four statements in Table 1-1.

Table 1-1. Basic operations

Operation	Example	Sample Command
Create	store(DocumentID, DocumentValue)	add(), set()
Read	get()	get(DocumentID)
Update	Update(DocumentID, DocumentValue)	replace(), incr(), decr(), append(), prepend(), cas()
Delete	delete(DocumentID)	delete()

As you can see, the basic operations are very simple, and this ultimately makes the programming and application development very simple. Documents are updated wholesale, when creating, you store the whole document, and when updating, you get the whole document, change the content or portion you want to change, and save it back.

Some additional information that apply to all the operations:

- It should be noted that all operations within Couchbase Server are atomic. That is, an operation either works, or it doesn't. The same operation sent by multiple clients will be processed sequentially in the order they were received.
- Related to the previous entry, for all operations the response is either success or failure. If you store a value, it will either succeed, or fail. If you get a value, you will either get the value, or it will fail (i.e., it doesn't exist). There are no mid-error points in the operations. That doesn't mean you don't need to handle errors; if there is an error during an update operation, for example, you will want to be able to recover from and resolve that problem. There are also some operational error conditions from the server that indicate a temporary issue.

These core operations will let you do 95% of the work and operations you would need to do with any database. But there are also some additional operations that provide some more advanced operations, or that provide additional information, such as the observe command.

The actual interface to Couchbase Server for these operations is supported by the memcached protocol. In fact, Couchbase Server is 100% memcached protocol compatible and you can use existing memcached compatible applications with Couchbase. You do, however, gain more functionality by using the full client library and taking advantage of the full cluster functionality.

Time to Live (TTL)

All values can be stored with an optional TTL, or Time to Live, value. The expiry time is particularly useful for data or information that is transient in nature, such as session stores for a website, or baskets during a shopping session. Identification of whether the item has 'expired' or not works through two mechanisms:

- Lazy identification at the point of access. If you try to to perform a GET or UPDATE or other operation on a value, and the value has expired when the operation occurs, the operation will proceed as if the previous document never existed.
- Background expiration operates on a schedule (default is hourly), and it deletes items from memory. This allows for the data to be be removed from the built-in caching layer and the version stored on disk. This deletes the item (and frees the RAM and disk space), regardless of whether or not the data has been accessed.

Data Consistency

One common question with Couchbase Server is how to ensure information is consistent across the cluster. Because of the structure of the server, there are no multiple copies

of the data. Because the information is automatically sharded across the cluster according to the hash of the document ID used to store the information, there is only one active location for a particular document.

All sets, gets, and updates on the document take place on the same server, in the single active record of the information. There are no consistency issues with the information spread across the database because of this single active copy architecture.

This architecture works on the individual document access model. There are additional layers to the system that may ultimately be affected in a different way by the consistency model. The views and indexing system (described later in this chapter), relies on a separate model when updating and building the index information, and the consistency of the information and the updates work in a very specific way.

From an operational level, you can also determine whether a particular document update has been persisted to replicas, and/or persisted to disk. If the document has been persisted to disk, then that means it's eligible for inclusion in the view indexes. All this information can be determine through the use of the observe command and, where supported by the client library, update operations that support durability requirements.

In Couchbase Server, developers are given a lot of choice about how these different components work, and how they operate together. There are choices and decisions that can be used to drive the operations, and to control the methods and systems used. The consistency of views and document updates can be controlled to achieve the result you need, but it may require a small tradeoff in performance.

Data Concurrency

The concurrency of access to the information is related to the fact that there is only one active version of the data. Operations on a document are atomic—that is, they either work or they don't. It is not possible to 'accidentally' corrupt the information by updating the record from two or more hosts simultaneously. Either the operation will succeed both times (sequentially), or one will succeed and the the other will raise an error condition that will then need to be handled. In reality, the chances of updating the same document simultaneously, given that most operations will complete in microseconds, is unlikely.

This still leaves the question of protecting the updates of information. Although you cannot corrupt the document by updating them at the same time, it is still possible to overwrite or update a document with a different version of the information. The Couchbase Server supports a number of operations that help with this, the primary one is Check-and-Set, or "Compare and Swap" (CAS), which adds a check value to the request so that you only update the when the check value supplied matches. The check value is updated every time the document changes, so if two clients obtain the record, and then

try to update it, the first one will work, but the second one will fail because the check values do not match (a CAS miss).

Other atomic operations include increment and decrement, and append and prepend which update the data on the server without having to perform a server/client/server roundtrip.

There are also explicit locking operations that will stop an item being updated until the lock is released. I'll describe these in more detail when we look at the specific operations and the client libraries and SDK interface.

Views, Indexes, and Querying

The power of a database doesn't come from the ability to store information using specific document IDs and associated content, although you can do an awful lot with just this information. You gain a lot by being able to search, query, and unique link your data that makes it powerful.

In Couchbase Server 2.0, a single system, Views, enables you to translate your document data (ideally stored in JSON) into a fixed format, and using that structured format, you can then query and search database for information. Because the view works by examining the schema-less document data, and converts that into a structured format, you can process and work with the information in a number of different ways. It also means that you can process different documents with different formats into a structured form. This can be particularly useful as your application matures and changes, because the views system can cope with different JSON documents as the schema changes without requiring you to change all your documents to match a new structure.

Views are written using map/reduce functions; the map performs the translation from the document structure to the table output structure. The reduce function can be used to simplify and summarize data, such as building counts, sum totals, or more complex condensations.

Traditional map/reduce is an expensive process, because normally you have to run the map/reduce on the entire dataset each time. Couchbase Server uses a system called incremental map/reduce. The basic map/reduce process creates an index, and the index can be incrementally updated as the source data changes. For example, if you have 10,000 records, and create a view on the information, then when you update 200 records, only those 200 need to be reprocessed for the index to be updated. This means that your views (and indexes) can be kept up to date with your underlying data according to the schedule you set, and much quicker than having to completely reprocess the entire dataset each time.

Once the view, and the corresponding index, have been created, the information can then be queried. In fact, the query mechanism it built on top of the map/reduce structure

that you create. Querying your data effectively is therefore a combination of understanding what you want to query, the underlying document structure, and writing a suitable view to produce the index you need to perform that query.

Views are a large topic, but once you get your head around the basics, they are very powerful. Views often confuse the traditional SQL users because they seem to work in reverse. In the SQL world you create the strict structure so that you can query it after the data has been stored and processed. With Couchbase, you store the data in any format and post-process it to extract the information you need in the format you need. It doesn't normally take long to see the benefits.

Comparing Couchbase to SQL Databases

Comparing the operation and development model of the SQL databases to Couchbase, and indeed document databases in general, is an entire topic and book on its own.

The key difference is the flexibility of the document structure. Document databases such as Couchbase Server allow information to be stored into the database in simple documents, rather than the rigid, schema enforced structure of databases, tables and fields.

Getting information into, and out of, Couchbase Server is generally much more straightforward, and documents can be modeled using more application friendly structures with the information grouped together. For example, with contact information, you can have a single record that covers an entire contact, including multiple email addresses, phone numbers, and other information. In an SQL database, you would either use a fixed table structure, which might limit the number of fields for a particular type of information, or you might use relations between tables to allow for unlimited entries. The latter sounds sensible, until you have to view the information as one record when it requires collecting the information from ten or twelve different tables.

Better still, the document structure allows for the structure to change over time. For example, 20 years ago including an email address for every contact was unnecessary, adding it to an SQL database would mean either adding more tables (and more code to be able to load them), or changing every existing record to add a new field.

This simplified structure offered by document databases not only makes the data architecture easier, it can also makes development easier, as the process of storing and retrieving information is simpler, enabling you to concentrate on the application functionality, and not the complexities of reading and writing to SQL databases.

Beyond the development advantages, it is the ease of scalability, which makes it straightforward to extend and expand your database as your application and data needs increase. Within a typical SQL environment this scalability is more difficult, and must be architected and incorporated into your application structure before you start adding the data. With Couchbase Server, the scalability is built into the database and your appli-

cation doesn't need to know about the complexities of the database architecture, and doesn't need to be changed whether you are running on one node, ten nodes or a hundred nodes.

Use Cases

Couchbase Server excels in a number of different environments. The high performance and concurrency aspects of the database make it ideal for those applications where you have a high number of users performing both reads and writes to the stored data.

For example, online and social gaming involves large numbers of users creating, updating and retrieving large volumes of information as they collect, use, and exchange objects. Session stores fall into this category too, where the consistent high performance and the ability to effectively red and write large volumes of information are key. Content provision, and ad-targeting are also quite popular, where the speed of access to the content, and the ability to track usage statistics of that information is very high.

With the views and querying mechanism, there are a huge range of different applications that can be written to take advantage of the high performance nature, and query ability. Couchbase Server can be used in situations where a MySQL database and memcached caching layer have previously been used together, but without the management complexities that this architecture normally involves. Because Couchbase Server is also memcached compliant, you can also use it as a direct memcached replacement within your existing database and application environment, and still take advantage of the scalability it provides.

CHAPTER 2

Getting Started

Before we start developing with Couchbase Server, you need to install the main server and a suitable SDK for your chosen language. The core interface and operations of storing, retrieving and updating data are the same.

Installing Couchbase Server

Installing Couchbase Server is a two phase process. First, you need to install the Couchbase Server software, then you need to perform a very simple setup procedure that configures some basic properties. The entire process is designed to be as quick and simple as possible, and should take you less than five minutes, depending on the speed of your internet connection.

Couchbase Server is available for the three main platforms:

Redhat-based and Ubuntu-based Linux
: Download the installer package and use *rpm* or *dpkg* to install the software.

Windows Server
: Download and double-click on the MSI installer package and follow the onscreen instructions.

Mac OS X (developer use only)
: Download the package and drag and drop the application to your Applications folder.

I recommend you check the "Getting Started" pages in the documentation to understand the precise platforms and other limitations. You'll also find more specific installation guides and information.

Once the software is installed, you'll need to perform the online setup process. This will ask you a number of important questions. You can always simply use the default options

listed; there are no inherent problems or issues with doing this, but a little preparation can go a long way to making the process easier later on.

The setup process can be completed in a number of different ways, including from the command-line and using the REST API, but the easiest way the first time you install is by using the web interface. Couchbase Server comes with an extensive web administration console (see Getting Started with Couchbase for more information), and we can use that console and interface to perform the basic setup steps.

You should consult the Couchbase Server Best Practices chapter for advice and guidance on the best way to deploy and use your Couchbase Server. You can find it at this page.

Before we start, you need to think about three key configuration points:

Location for persistent storage
: Data in Couchbase Server is persisted to disk, so you need to have a directory where you can store that data. On Linux and Windows, the default location is within the directory structure for the installed software. On Mac OS X, the default location is within the Application Support directory of the user that executed the software.

Server Quota
: Data in Couchbase Server is cached, and the configuration of the RAM allocated to that cache is configured in two levels. The first is the server quota, which is the total amount of RAM allocated to the cache on the server. The server quota configuration is shared by all the nodes within the cluster. So if you allocate 2GB of RAM on your first node, and then add three more nodes, you will have 8GB RAM cache allocated (four nodes * 2GB each).

  The Server Quota RAM allocation is used while Couchbase Server is running, so if you have other applications running you may want to reduce the total of memory allocated to Couchbase Server. Especially if this is your development machine, you may not want a large portion of RAM allocated. When you deploy your application, you will want as much RAM as possible to ensure the highest performance.

  The quota is just that; a quota. The figure you configure here is the RAM used for the caching layer. The application can store more data than the configured quota; Couchbase Server will automatically handle the storage and loading of information from disk, caching the most frequently used items in the configured RAM quota. There is, for obvious reasons, a performance hit when loading information from disk over a cached record.

Bucket Quota

During setup, you will create a "default" bucket. Buckets are designed as containers for a particular group of data or application, and you can have multiple buckets within your cluster. For example, you might have a bucket to store session information, and another bucket to store recipe data. Bucket quotas exist to allow you to control how much of the server quota is allocated to a given bucket, and can therefore be used to ensure that buckets with the highest performance need are given a greater proportion of the RAM cache.

We'll quickly walk through the setup process, and the main questions and issues raised, start by opening a web browser and pointing at port 8091 on the host running Couchbase Server. If that's the same machine, the URL would be *http://localhost:8091/*. You should get a screen like the one shown in Figure 2-1.

Figure 2-1. Setting up the Couchbase Server

The following steps will help you get set up.

1. Click Setup to get started.
2. Choose a location for the disk storage. The database and index paths are configured individually, and best practice is to set them to different disks to get the best performance. Since you are creating a new cluster, leave the default setting; but you could add a new node to join an existing cluster now if you had one.

   Set the per server RAM quota. Remember, this setting will be used for all the nodes in your cluster. You can change this allocation at a later date, but it's a good idea to pick a suitable value. Remember that this is the amount of memory that will be available for caching data within Couchbase Server.

   Click Next when you are ready.
3. Choose if you want to load the sample databases. A number of sample databases are included by default with Couchbase Server and include both documents and views. They can be a useful way to learn about best practices for designing the data and your document structure. Installing the sample data may take some time, so be patient. Click Next when ready.
4. You will be asked to create a 'default' bucket. The default bucket provides you with an initial storage location for data, but you should not use it for anything but testing. When building your own application, and particularly during deployment, you should create a bucket specifically for your application. You can see the window shown in Figure 2-2.

   Buckets can be of two types, either memcached (cached only), or Couchbase (persisted and scalable). For best results you should use the Couchbase bucket types. A brief rundown of the configuration available is:

   Memory Size
   : This option specifies the amount of available RAM configured on this server which should be allocated to the default bucket.

   Replicas
   : For Couchbase buckets you can enable replication to support multiple replicas of the default bucket across the servers within the cluster. You can configure up to three replicas.

     You can disable replication by deselecting the Enable checkbox. You can also select whether to create replicas of the index data automatically.

     To configure the number of replica copies, select the number of replicas using the Number of replica (backup) copies popup list.

     You can safely use the defaults—no replicas will be created until you have enough nodes, so there is no performance or other impact.

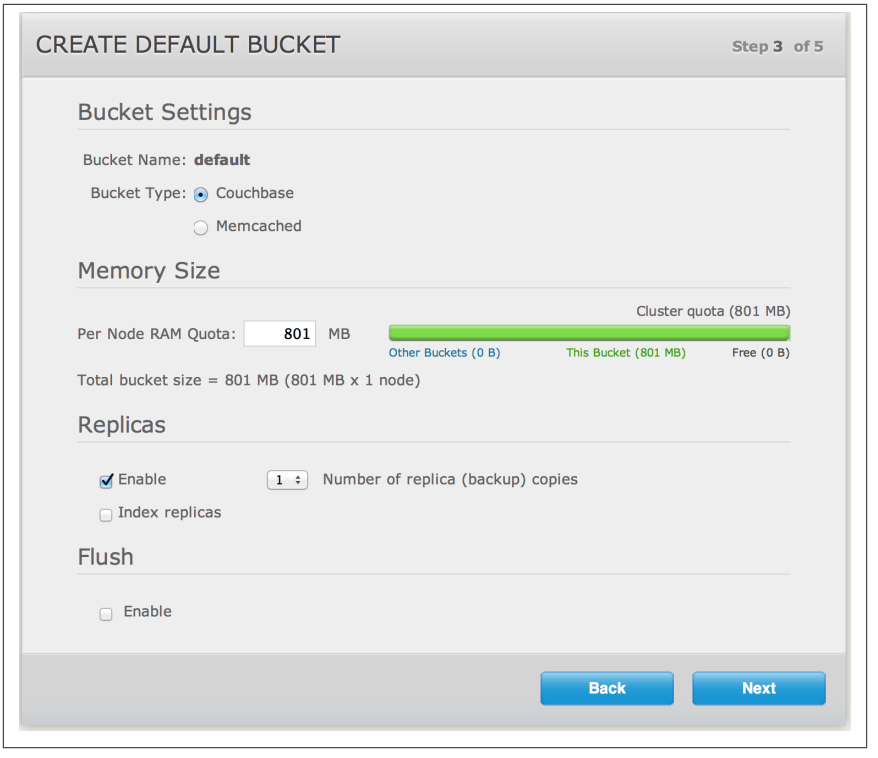


Figure 2-2. Creating default bucket

Flush
: Enable or disable support for the Flush command, which deletes all the data in an a bucket. The default is for the operation to be disabled. Flush can be useful during development as it enables you to delete bucket data and start again. To enable check the Enable checkbox.

Auto-compaction
: Persisted data and view indexes are stored on disk, but over time the stored information can become fragmented. An automated compaction process will recover the fragmented disk space for you according to some rules. You can leave the defaults in place, but when deploying an application within Couchbase Server you may want to alter the settings. Check the documentation for more information.

In all cases the default values can be used. You may want to reduce the bucket size so that there is RAM quota available for creating your own application-specific buckets. Click Next to continue.

5. The final step is to set an administration password to be used when accessing the server. Note that this is the administration password, and is not required for accessing data.

Once everything is completed, you should be presented with the Couchbase Server Cluster Overview screen, as seen here in Figure 2-3.

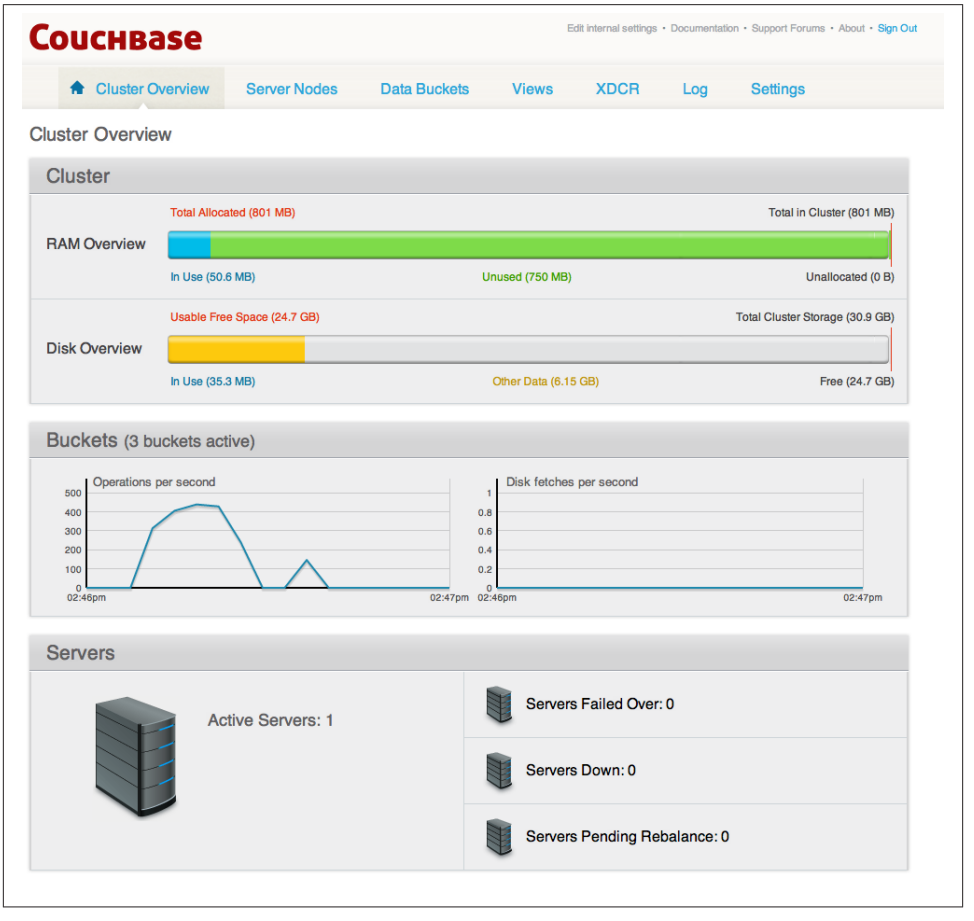


Figure 2-3. Couchbase Server Ready!

This means your Couchbase Server is ready to use!

Couchbase Client Libraries

For the best results, you should use the Couchbase client libraries rather than a memcached library. These include support for all of the core operations, an interface to the

Views system for querying the database, and can communicate with the cluster as a whole, automatically updating their configuration as the structure of the Couchbase Server cluster changes.

There are clients available for:

- Java
- .NET
- Ruby
- Python
- PHP
- C

Some third parties are working interfaces for Perl and Node.js. Installation is generally very straightforward for each library, pick your chosen library and use the notes below to get started.

Under the hood, Couchbase Sever is based on the memcached protocol with some additional extensions. This means that you can use any memcached compatible client to talk to Couchbase Server. This will limit the available operations and enhanced support for clustering such as the observe command.

The latest code and documentation can always be found on the Couchbase website.

Java

The Java Couchbase Client Library can be installed either by downloading the JARs, and then including them in your normal project and build processes, or by using Maven.

Once you have downloaded the Jar, you will need to make sure it is included in your builds and deployments. For example, from the command-line:

```
shell>     javac     -cp     couchbase-client-1.1.jar:spymemcached-2.8.4.jar
\

Main.java

shell>      java      -cp      .:couchbase-client-1.1.jar:spymemcached-2.8.4.jar:
\
jettison-1.1.jar:netty-3.3.1.Final.jar:commons-codec-1.5.jar:
\
httpcore-4.1.1.jar:httpcore-nio-4.1.1.jar Main
```

If you are using Ant, Eclipse or similar, you can explicitly add the library to your project.

.NET

The .NET library can either be downloaded as a Zip file or run the following in the NuGet Package Manger console:

```
PM> Install-Package CouchbaseNetClient
```

To build an application using the library, you'll need to add the *Couchbase.dll* and the *Enyim.Memcached.dll* assemblies to your project within Visual Studio. You will also need to ensure that your project is linking against the full .NET Framework, rather than the default .NET Framework Client Profile.

Python

The easiest way to get the Couchbase Python Client Library is to use PIP.

```
shell> pip install couchbase
```

To use the library once installed, you should import the client components from the install `Couchbase` module. For example:

```
#!/usr/bin/env python

from couchbase.client import Couchbase
```

The client group includes everything you need for basic Couchbase operations.

Ruby

The best way to install the Ruby client library is to use Rubygems. In order to use the library, your Ruby version should be 1.8.7 or higher and you must have Rubygems 1.3.6 or higher. There are some prerequisites to the installation:

1. Install the package from *libevent*.
2. Then install the Couchbase C Client library, from *libcouchbase*.
3. Install the *libvbucket* library which is also part of the Couchbase C client library.

4. Instll the `couchbase` Ruby gem:

```
shell> gem install couchbase
```

To use, you must import both the `rubygems` and `couchbase` modules into your Ruby scripts. For example:

```
require 'rubygems'
require 'couchbase'
```

Couchbase functionality is exposed through the `Couchbase` class.

PHP

The PHP client library is provided as a standard PHP extension that you can install along with other extensions. To do this:

1. Download the SDK for your system from the SDK page.
2. Unpack the archive containing the SDK, which is supplied in specific architecture and operating system packages:

   ```
   shell> tar xzf php-ext-couchbase-$system-$arch.tar.gz
   ```

   The resulting directory includes a file *couchbase.so* which is the PHP extension for the SDK. Note the path to this file.
3. Find the *php.ini* file for your PHP interpreter. Try:

   ```
   shell> phpi -i | grep ini
   ```

   The output provided will have configuration file that is loaded for PHP, for instance:

   ```
   Loaded Configuration File => /private/etc/php.ini
   ```
4. Open the php.ini file and add the path to your couchbase.so file. For example:

   ```
   extension=/path/to/couchbase.so
   ```

   You may also need to download and install the JSON extension, which you must also add to your *php.ini* file:

   ```
   extension=/path/to/json.so
   ```

   Couchbase functionality is exposed through the `Couchbase` class.

Buckets

Data is stored within Couchbase Server in logical units called Buckets. Buckets are individually isolated from each other, and you can configure the connectivity to the bucket to require a password (using SASL authentication). As we've already seen, buckets also have their own RAM quota and replica settings.

Connecting to a Cluster

Connecting your application to a Couchbase cluster is fundamentally the same in every language; you have to provide the IP address or hostname of just one node within the cluster. Once that initial connection has been made, the client retrieves a "cluster map" that tells the client library about all of the nodes in the cluster and their IP addresses. Communication is then handled directly to the nodes within the cluster.

The basic connection process is to provide the URL of the node you have chosen in the cluster. The URL is of the format `http://hostname:8091/pools`. This is actually a REST API endpoint that provides the configuration information about the cluster. The `pools` path is where that configuration information lives on the server.

Depending on the library, you must then specify the bucket (and password, if it requires one) when opening the connection. For example, within Java you would use the following fragment to connect to the server:

```
List<URI> uris = new LinkedList<URI>();

// Connect to localhost or to the appropriate URI
uris.add(URI.create("http://127.0.0.1:8091/pools"));

CouchbaseClient client = null;
try {
  client = new CouchbaseClient(uris, "default", "");
} catch (Exception e) {
  System.err.println("Error connecting to Couchbase: "
    + e.getMessage());
  System.exit(0);
}
```

In the above, we are connecting to the bucket called "`default`" (the second argument when creating the client object). To connect to a different bucket, just specify a different name.

In Ruby, the bucket name that you want to connect to is appended to the URL of the server, but you are still creating client object to form the basis of your interface:

```
client = Couchbase.connect("http://127.0.0.1:8091/pools/default")
```

To connect to a different bucket:

```
client = Couchbase.connect(:host => "http://127.0.0.1:8091/pools/default"
                           :bucket => "beer-sample")
```

To connect to a bucket that has authentication and a password enabled, the accepted format is use a username/password combination. Couchbase Server doesn't support multiple users, so the username is the name of the bucket. For example:

```
Couchbase.connect('http://localhost:8091/pools/default/buckets/recipes',
                  :username => 'recipes',
                  :password => 'secret')
```

The problem with this solution is that you are connecting to only one host. Although the client libraries are intelligent and will get the entire cluster configuration and then talk directly to individual nodes, if the first node you talk to happens to be down, you will not be able to talk to the cluster at all. In Java, you can handle this by adding more nodes to the list of URIs, like so:

```
List<URI> uris = new LinkedList<URI>();

// Connect to localhost or to the appropriate URI
uris.add(URI.create("http://192.168.0.60:8091/pools"));
uris.add(URI.create("http://192.168.0.65:8091/pools"));
uris.add(URI.create("http://192.168.0.78:8091/pools"));
```

In .NET, you specify the list of URIs as part of the settings in *App.config*:

```
<servers bucket="private" bucketPassword="private">
      <add uri="http://10.0.0.33:8091/pools/default"/>
      <add uri="http://10.0.0.34:8091/pools/default"/>
</servers>
```

In other languages, you may need to manually iterate over a list of servers to open the initial connection. For example, in PHP you might create an array and iterate over it:

```
$servers = array("192.168.0.72:8091","127.0.0.1:8091");

foreach($servers as $server) {
  $cb = new Couchbase($server, "", "", "default");
  if ($cb) {
    echo "Connected to $server";
    break;
  }
}
```

Check the documentation for the language you want to use to check for built-in multi-host support.

CHAPTER 3

Document-based Database Design

Couchbase Server is strictly a document database, and as we've seen you store data into the database by using a document ID and the corresponding document data. That data can be in any format, and indeed there is nothing to stop you using a serialized object or stream of raw bytes and storing and retrieving this information.

With that in mind, you will get the best benefits out of Couchbase and the clustered architecture if you use Couchbase to store and use the data that is most frequently used and updated by your application. The low latency on reads and writes that Couchbase Server provides makes it ideal for the highly interactive portions of your application, rather than as a generalised data store for both active and historical information.

Modelling data in documents and Couchbase Server is about understanding how to store and reference different types of information. For example, some data is easily identifiable as a simple field. Other data is more usable and accessible as an array or object style within the JSON structure. We'll start by looking at JSON before building a sample model based on our recipe database.

JSON Overview

JSON is a lightweight, easily parsed, cross-platform data representation format. There are a multitude of libraries and tools designed to help developers work efficiently with data represented in JSON format, on every platform and every conceivable language and application framework, including, of course, most web browsers and the JavaScript language supported within them.

JSON supports the same basic value and variable types as supported by JavaScript. These are:

Number
: A number can be either integer or floating-point.

JavaScript supports a maximum numerical value of 2^{53}. If you are working with numbers larger than this from within your client library environment (for example, 64-bit numbers), you must store the value as a string.

String
: A string should be enclosed by doublequotes and supports Unicode characters and backslash escaping. For example:

```
"A String"
```

Boolean
: A boolean value is either `true` or `false`. You can use these strings directly. For example:

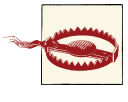
```
{ "value": true}
```

Array
: An array is a list of values enclosed in square brackets. For example:

```
["one", "two", "three"]
```

Object
: An object is a set of key/value pairs (i.e., an associative array, or hash). The key must be a string, but the value can be any of the supported JSON values. For example:

```
{
      "servings" : 4,
      "subtitle" : "Easy to make in advance, and then cook when ready",
      "cooktime" : 60,
      "title" : "Chicken Coriander"
}
```



Because of the significance of JSON to the server, documents submitted to the system are checked to determine if the information is valid JSON or not. This information is used to update the metadata about the document, which you can then use during view processing to only index JSON documents if you are mixing document types in your datastore.

Modeling Recipe Data

To provide a basis for understanding the document structure and how it can be used, let's start by looking at a sample Couchbase Server database built on top of recipe data.

Recipes contain a whole host of different information, from the core data, such as the recipe title, to keyword data like the cuisine type, to more complex ingredient information and a simpler list of individual method steps. Let's start with the basic data.

Core Data

The core information about a recipe can be stored as fields within the top level of the JSON structure. For example, we can store the title and a subtitle to hold a more detailed description of our recipe. We'll also put a number of servings in there:

```
{
  "title": "Amorous chicken with grapes in a seductively creamy wine sauce",
  "subtitle": "Serve with new potatoes and sugar snap peas.",
  "servings": "2"
}
```

We can also put other information into this top level that relates to the recipe, such as cooking time. Actually, cooking time is often divided into a number of different elements, such as the preparation time and the actual time spent cooking,

```
{
  "title": "Amorous chicken with grapes in a seductively creamy wine sauce",
  "subtitle": "Serve with new potatoes and sugar snap peas.",
  "servings": "2",
  "preptime": "20",
  "cooktime": "13",
  "totaltime": "33"
}
```

We've now got a basic recipe structure. Looking forward a little, it should be possible to get out a list of recipes, sort by the time it takes to cook them, and allow selection by recipe title information.

Keywords or Tags

Keywords or tags are used against all sorts of different data so that you can classify them. Within a document database, a good way of tracking this information is with an object or hash style reference. That makes it really easy to pick out the information just by accessing the key directly. For example:

```
{
  "title": "Amorous chicken with grapes in a seductively creamy wine sauce",
...
"keywords": {
    "diet@peanut-free": 1,
    "cook method.hob, oven, grill@hob": 1,
    "diet@corn-free": 1,
    "diet@citrus-free": 1,
    "occasion@entertaining": 1,
    "diet@shellfish-free": 1,
    "main ingredient@poultry": 1,
    "diet@demi-veg": 1,
    "cuisine@european.french": 1,
    "meal type@main": 1,
    "diet@egg-free": 1,
```

```
      "occasion@valentines": 1
    },
  }
```

Using this method we can easily look directly for a given keyword by looking up its object key (`diet@corn-free`), rather than iterating over the array. This can be really useful within your application level.

Ingredients

Recipes are nothing without a list of ingredients to work with. There are potentially lots of pieces of information that we could incorporate, but let's look at three critical pieces of information, the raw ingredient, the ingredient as it might be displayed in the recipe, and the measurement:

```
    {
      "ingredtext": "seedless white grapes, halved",
      "ingredient": "seedless white grapes",
      "meastext": "25 g"
    },
```

Using a hash-style for each ingredient means that we can easily identify each component in the ingredient set.

In a recipe, we'll have a list of ingredients. It's unlikely we'll want to refer to the ingredients except as the full list, so that can be describe as an array of these individual ingredient records:

```
{
  "title": "Amorous chicken with grapes in a seductively creamy wine sauce",
  "preptime": "20",
  "servings": "2",
  "subtitle": "Serve with new potatoes and sugar snap peas.",
  "totaltime": "33",
  "cooktime": "13",
  "ingredients": [
    {
      "ingredtext": "seedless white grapes, halved",
      "ingredient": "seedless white grapes",
      "meastext": "25 g"
    },
    {
      "ingredtext": "double cream",
      "ingredient": "double cream",
      "meastext": "50 ml"
    },
    {
      "ingredtext": "dry white wine",
      "ingredient": "dry white wine",
      "meastext": "125 ml"
    },
```

```
    {
      "ingredtext": "olive oil",
      "ingredient": "olive oil",
      "meastext": "3 tsp"
    },
    {
      "ingredtext": "plain flour",
      "ingredient": "plain flour",
      "meastext": "3 tsp"
    },
    {
      "ingredtext": "seasoning",
      "ingredient": "seasoning",
      "meastext": ""
    },
    {
      "ingredtext": "seedless red grapes, halved",
      "ingredient": "seedless red grapes",
      "meastext": "25 g"
    },
    {
       "ingredtext": "skinless boneless chicken breasts, trimmed and cut into
strips or cubes",
      "ingredient": "chicken breasts",
      "meastext": "2"
    },
    {
      "ingredtext": "butter",
      "ingredient": "butter",
      "meastext": "25 g"
    }
  ]
}
```

The limitation of this array of ingredient items is that it does make it more complex to update individual items, but, as we are dealing with whole documents each time, when an individual ingredient is updated, the entire document and structure will be updated at the same time.

Methods

For the method data there are a variety of techniques we could use. Methods in recipes are normally straightforward lists of tasks to complete, so we could use an array of the method steps, like this:

```
{
...
   "method": [
       "Melt the butter in a heavy based frying pan with the oil. Add the chick-
en breasts and saute for 10-15 min, turning frequently until cooked and golden.
",
```

```
        "Remove the chicken from the pan with a slotted spoon. Keep warm while
making the sauce.",
        "Stir the flour into the remaining butter and oil in the pan and cook
for 1 min. Gradually whisk in the wine until all is incorporated and bring to a
simmer. Cook for 1 min or until thickened. Remove from the heat. ",
         "Gradually stir the cream into the sauce with the grapes. Season to
taste. Simmer together gently for 1-2 min or until piping hot.",
         "Cut the chicken breasts into thick slices diagonally and arrange on
warmed serving plates. ",
       "Pour the sauce over the chicken and serve."
   ]
}
```

The limitation of this structure are that we have no way to divide the information between different parts. For example, if your recipe has both a sauce and a main component, the method doesn't distinguish between these two sets of methods.

Related Data

Very rarely does data generally stand on its own. With a recipe, for example, we probably have a bunch of related information. In the background, for example, we might have some nutritional information, such as the calories, fat, and carbohydrate content. We could add that into the recipe record:

```
{
  "title": "Amorous chicken with grapes in a seductively creamy wine sauce",
  "subtitle": "Serve with new potatoes and sugar snap peas.",
...
  "nutrition": {
     "fat" : 9.8,
     "calories": 128,
     ...
     }
}
```

Remember that in Couchbase Server, the time required to load the data, assuming it's stored in RAM, is just the latency over the network. Adding a small amount of nutritional information might not add much to the process. However, if it's infrequently used information, sometimes it can make more sense to store a different document, but with a related ID. For example, we could create a document with a suffix of the recipe's document ID with the nutritional information.

There are no explicit, or enforced, relationships within Couchbase Server, but you can add fields to one document to help identify where other information might be stored. For example. we could add a field to the main recipe document with the ID of the document containing the nutrition data:

```
{
  "title": "Amorous chicken with grapes in a seductively creamy wine sauce",
  "subtitle": "Serve with new potatoes and sugar snap peas.",
  "servings": "2",
  ...
  "nutrition_data": "AD3D63D6-2FE0-11E2-93F9-898688C79046_nutrition"
}
```

If you use this method to create multiple different types of data in your database, another good practice is to ensure that you store the document type in the document data. This will make it easier when creating views on the information. It can also be useful to record a version number. This allows you to track the version of the document structure, so that if the structure changes later we can identify that in the application and either upgrade the data, or handle it differently. This changes our document to:

```
{
  "title": "Amorous chicken with grapes in a seductively creamy wine sauce",
  "subtitle": "Serve with new potatoes and sugar snap peas.",
  "servings": "2",
  "doctype" : "recipe",
  "formatver": "1",
  ...
  "nutrition_data": "AD3D63D6-2FE0-11E2-93F9-898688C79046_nutrition"
}
```

The nutrition data was a one to one relationship. Other data types are one-to-many. For example, you want to attach comments to your recipes. One way to handle this relationship is to ensure that the comment record has the recipe document ID attached, like this:

```
{
    "user": "mc",
    "comment": "delicious",
    "doctype": "comment",
    "recipe": "AD3D63D6-2FE0-11E2-93F9-898688C79046"
}
```

When outputting the information from a view, we can use that to output the comment information by performing a search on all the comments that contain the document ID reference.

CHAPTER 4
Getting Data In and Out

With Couchbase Server installed, and your chosen client library up and running, it is time to start storing and retrieving information into the database. All of the client libraries use the same basic operations, although they may have different function or methods names. Despite the differences that exist between individual client libraries, the basic operations and how and why you use them are the same.

A basic list of the key Create, Retrieve, Update, and Delete (CRUD) operations is shown in Table 4-1.

Table 4-1. Key CRUD operations

Operation	Description
`add(id, document [, expiry])`	Add an item if ID doesn't already exist
`set(id, document [, expiry])`	Store a document with ID
`replace(id, document [, expiry])`	Update the document for an existing ID
`cas(id, document, check [, expiry])`	Update the document for an existing ID providing the check matches
`get(id)`	Get the specified document
`incr(id [, offset])`	Increment the document value by offset
`decr(id [, offset]`	Decrement the document value by offset
`append(id, value)`	Append the content to the end of the current document
`prepend(id, value)`	Prepend the content to the beginning of the current document
`delete(id)`	Delete the specified document

There are, though, common use cases and sequences with the different operations that can be used to achieve different processes and solutions. Let's look at some of these in the context of a typical web application.

Basic Interface

To start, let's look at a basic Ruby script that stores some data, and retrieves it, and handles the basic responses. A sample program, *hello-world.rb*, is shown in Example 4-1.

Example 4-1. Hello World!

```
require 'rubygems'
require 'couchbase'

client = Couchbase.new "http://127.0.0.1:8091/pools/default"
client.quiet = false
begin
  spoon = client.get "spoon"
  puts spoon
rescue Couchbase::Error::NotFound => e
  puts "There is no spoon."
  client.set "spoon", "Hello World!", :ttl => 10
end
```

Dissecting the script reveals the `set` and `get` process common in Couchbase:

- The first two lines load the necessary libraries.
- The next line opens up a connection to your Couchbase Server cluster. Remember that you need to connect to only one node in the cluster. There could be 20 or just one node in the cluster we are connecting to in the example.

  The definition is through a URL which should point to at least one node within your cluster. In this example, the `localhost` address is used. The remainder of the URL is the pools definition which remains the same.
- The remainder of the script performs a retrieve and store operation. If the initial retrieve operation (for the document ID `"spoon"`) fails, then we set the data into the database. If the document ID does exist, the script prints out the stored value.

When connecting to the "`default`" bucket, you do not need to supply any credentials. For all other buckets, you must provide the username (the bucketname), even if no password is specified.

You can test this script out by running it from the command line. The first time you run it, it should output this error string:

```
shell> ruby hello-world.rb
There is no spoon.
```

The specified document does not exist in the database, but is added after the error string has been printed. The second time you run it, you should get the stored document value:

```
shell> ruby hello-world.rb
Hello World!
```

As an additional demonstration, the welcome string stored has been given an expiry value of 10 seconds. This means that if you wait longer than 10 seconds after you have stored the value, the value will be deleted from the database. If you wait more than 10 seconds from the first time you ran the script and execute the script again, it should output this error string:

```
shell> ruby hello-world.rb
There is no spoon.
```

For comparison, the same script, written in PHP:

```
<?php

$cb = new Couchbase("127.0.0.1:8091", "", "", "default");

$spoon = $cb->get("spoon");

if ($spoon) {
  echo "$spoon";
}
else {
  echo "There is no spoon.";
  $cb->set("spoon", "Hello World!", 10);
}

?>
```

Although this is a very basic example, it demonstrates the simplicity of retrieving and storing information into Couchbase Server, and the expiry of information.

The basic sequence shown here is one that will be replicated in many different places in your code, and it's important to note the simplicity of the document-based interface. Note that there are no complex SQL statements to write, no need to worry about creating a structure to hold the data, or any need to worry about the size and complexity of the cluster holding the information.

Now let's start looking at some more specific examples of different areas of the Couchbase Server interface.

Document Identifiers

Data is stored by recording a block of data against a given document ID. Because the document ID is the primary method for retrieving, and updating, the information that you store, some care needs to be taken to choose your document ID.

The document ID is required, and Couchbase will not automatically create one for you if an explicit ID is not specified. It should go without saying that document IDs should

also be unique; you can only retrieve a single document back from a given ID. Use the same ID and you will overwrite the existing information.

There are a number of different strategies available:

Convert a field to a unique ID
: Certain fields or values make good document IDs. For example, if you are storing session data, or even user data, you can use the unique session ID or the user's email address as a suitable document ID. For other data, such as a user's name (there are numerous Martin Browns out there), or recipe name (multiple varieties of Lasagne) a different ID structure should be used.

Store objects and sequences
: You can make use of the raw numerical storage and the increment operation to create a sequence ID. Create a document, "recipe-sequence" and store a bare integer. Each time you create a new recipe, increment the value, and append that to the end of a string, such as "`recipe_`". Because the increment operation is atomic, the number will increment and should never be repeated.

Use a UUID
: There are numerous UUID solutions available in different languages that can be used to create unique identifier for the purposes for storing a document. UUIDs make the process of storing the information very simple and straightforward.

Of course, there is nothing to stop you using combinations or basic elements of all three. For example, you might store global data into "named" documents that you can easily identify and explicitly reference in your code, while storing user data in sequenced documents.

Another best practice approach, regardless of which document ID structure you use, is to either prefix your document ID for type. For example having `user_98475894756` and `object_3978547645`. When storing JSON, consider using a `type` field as well to make it clear what the document type is.

Be careful with the length of your document IDs. All document metadata, including the document ID, is stored in memory at all times, so longer IDs make for a larger RAM footprint, which reduces the RAM available for caching the document data.

One area to be careful of is how you split up and define the document and data. We look at that in more detail in Chapter 3.

Time to Live (TTL)

As we saw in the opening example, you can store documents with an expiry time. All values stored within Couchbase Server have an optional expiry, or Time to Live (TTL) value. The expiry is designed for use on data that has a natural lifespan. For example, session data, or shopping baskets. The expiry time is configured using a simple numerical value and the value is interpreted differently according to its size:

- Any value smaller than 30 days (i.e., 30 * 24 * 60 * 60) is taken as a relative value in seconds. For example, storing a document with an expiry of 600 will expire the document in 10 minutes.
- Any value larger than this is taken as an absolute value from the epoch, also in seconds. For example, 1381921696 would be 16th October 2013.
- A value of zero (or not supplying an expiry time), means that the document has no expiry. The document will need to be explicitly deleted to be removed from the database.

Setting the expiry time can be performed when initially storing the value, and updating it. For convenience, you can also get the value and "touch" the expiry to update it. This is useful for session and other information where retrieving the record also implies the data should stay around longer. You can also explicitly touch (without a get) the expiry on a document.

Storing Data

To start with, you need to store data into your database so that you can later retrieve it. There are two primary storage methods:

- `set(docid, docdata [, expiry])`

  Store the specified document data against the document ID. This operation is an explicit set—that is, it will always store the document data against the ID you supply, irrespective of whether the document ID already exists, or what the corresponding value is.

  For example:

  ```
  $cb->set('message', 'Hello World!');
  ```

  This sets the document 'message' to 'Hello World!'.

  The expiry is an optional value, although different client libraries expose this in different ways. In PHP it's an optional argument which can be added to the end of our method call. For example, to add a ten second expiry:

  ```
  $cb->set('message', 'Hello World!', 10);
  ```

You can test the success of the operation by checking the return value in PHP:

```
if ($cb->set("spoon", "Hello World!", 10)) {
  echo "Message stored!";
}
```

In languages that support exceptions (Python, Ruby, .NET and Java) the exception system provides the information about whether individual operations succeed or note.

- `add(docid, docdata [, expiry])`

  The `add()` function works different to `set()`. With `add()` the operation will fail if the document ID you specify already exists. For example, in the code below, the first operation will complete successfully, the second will fail:

  ```
  $cb->add('message', 'Hello World!');
  $cb->add('message', 'I pushed the button, but nothing happened!');
  ```

  The `add()` function is useful when you are storing data into the database for the first time. For example, consider storing user data using the email address as the document ID. Using `set()` would overwrite an old user record. Using `add()` would fail the registration and indicate the user should use the password recovery system.

  The `add()` function is also atomic, so multiple writers can be adding data to the cluster at the same time, and it's safe to use within a multi-threaded environment.

Regardless of how you store the data, the actual format of the information you are storing is also important. For more details on designing data structures for document databases, see Chapter 3.

Retrieving Data

To retrieve a document from the database, you need only supply the document ID of the document that you want to retrieve. The retrieve operations are designed so that the `get()` operation will return an error if the corresponding document ID does not exist.

There is only one main function, `get()`. For example:

```
$message = $cb->get('message')
```

If the value comes back as undefined in PHP, then the specified document ID did not exist.

Retrieving in Bulk

There are times when you will have multiple different documents to collect. For example, if you've made use of Views, then you might have identified a number of documents that you want to retrieve. If you are using links within a document to other documents, you might want to load all of them at the same time.

As a rule, loading multiple documents in bulk is always faster and more efficient than loading the documents individually, largely because the latency in the requires and response has been removed by streaming the response of multiple documents.

Different libraries implement and expose this functionality in different ways, but the result is generally the same; either an array of the documents that you requested, or if supported by the language, a hash of the document IDs and corresponding document data. For example, in PHP the function is:

```
$ret = $cb->getMulti(array('recipe1','recipe2'));
```

This returns a PHP associative array, with each key/value pair as the document ID and document data. The document data is undefined if the requested document could not be retrieved.

Updating Data

Updating data in Couchbase Server should be performed carefully. You can only update entire documents, and because of the basic structure—updating document data against a document ID—the format is similar to that used when initially storing a document. In fact, you can use the `set()` function mentioned earlier. The problem is that you may not want to update a document that doesn't already exist. For this, you can use the `replace()` method. This only updates the document data if the specified document ID already exists. For example:

```
$cb->replace("welcome-message", "Hello World!");
```

The above will fail until we use either `add()` or `set()` to create the document ID.

Generally the process for updating information is to load the record, update the information in it, and then use `replace()` or `set()` to save it back again:

```
$message = $cb->get('message');
$message = $message . " How are you today?";
$cb->replace('message',$message);
```

When updating JSON-based documents, the operation is the same; you must load the existing document before updating a field and saving it back:

```
$record = array('name' => 'MC Brown', 'company' => 'Couchbase');
$cb->set('user', json_encode($record));

$newrecord = json_decode($cb->get('user'), true);
$newrecord["nickname"] = "MC";

$cb->replace('user', json_encode($newrecord));
```

To ensure you are not overwriting data that might have been updated by another client, you should use the CAS operation.

The above example uses the built-in `json_encode()` and `json_decode()` functions to serialize an internal associative array within PHP into JSON for storage and back again.

Concurrent Updates

In any highly concurrent environment, particularly many modern websites, you must make sure that you do not try and update the same information from two or more clients simultaneously. This is even more important in an environment such as Couchbase Server where updates occur very quickly, and where you are storing larger documents that may contain a lot of compound information.

For example, consider the following scenario:

1. Client A gets the value for the document "Martin".
2. Client B gets the value for the document "Martin".
3. Client A adds information to the document value and updates it.
4. Client B adds information to the document value and updates it.

In the above sequence, the update by Client B will overwrite the information in the database, removing the data that Client A added.

To provide a solution to this, you can use the compare and swap (`cas()`) function, which uses an additional check value to ensure that the version of the document retrieved is the same as the one currently stored on the server.

The result is a change to the above sequence:

1. Client A gets the value for the document "Martin" and the CAS ID.
2. Client B gets the value for the document "Martin" and the CAS ID.
3. Client A adds information to the document value and updates it, using the CAS ID as a check. The document is updated.
4. Client B adds information to the document value and tries to update it using the CAS ID. The operation fails, because the cached CAS ID on client B is now different from the CAS ID on the server after the update by client A.

CAS therefore supports an additional level of checking and verifies that the information you are updating matches the copy of the information you originally retrieved. CAS enforces what Couchbase calls optimistic locking, that is, we hope that we are the only

client performing an update that has the right CAS value, and that all clients always use a CAS function to do updates.

Within your code, CAS is a function just like the `update()` function. Depending on your environment, you may need to use a special get function (`gets()`) that obtains both the document information and CAS value.

For example, within Java you would update an existing document through CAS first by getting the value and stored CAS value, and then using the `cas()` method to update the document:

```
$value = client->get("customer", NULL, $casvalue);
$response = client->cas($casvalue, "customer", "new string value");
```

The limitation of using CAS is that it is not enforceable at a database level. If you want to use it for all the update operations, you must explicitly use it over the standard document update functions across your entire application. Care should be taken to use the operation in the right place. A CAS update is slower than a simple set operation, and to get the best performance you should use the fastest operation appropriate to your application and update requirements.

Server-side Updates

In addition to all the above operations for storing and retrieving information, there are a small number of server-side operations that update the stored data. They cannot be used with JSON documents, because the document information is not parsed, but they can be used for those documents where you are storing raw string or integer data. Here are some:

Increment
: When an integer value has been stored, it increments the stored value, either by one, or by the specific increment value. This is particularly useful if you are storing counters (for example, a scorecard, or a count of the number of visitors to a website or object), or using the values for sequences within the code.

  For example, to increment by one:

```
$cb->set('counter',10);
      $cb->increment('counter');
```

  To increment by 10:

```
$cb->set('counter',10);
      $cb->increment('counter', 10);
```

  Counters like this are still supported by the view system for indexing the data by using the document metadata. We'll look at this in more detail when writing some sample views.

Decrement
: When an integer value has been stored, it decrements the stored value, either by one, or by the specific increment value.

Append
: Append data to the end of the stored document. Useful for updating strings, or even text-delimited arrays of information. For example:

```
$cb->set('message','Hello');
      $cb->append('message', ' World!');
```

Will populate the typical message. For lists, you can append a fixed formatted string. For example:

```
$cb->set('userlist','martin,');
$cb->append('userlist', 'stuart,');
$cb->append('userlist', 'sharon,');
```

To get a list of users, you can access the 'userlist' record, and split them by commas, and ignore the blank last item.

Prepend
: Prepend data to the beginning of the stored document.

Remember, these operations are atomic, i.e., they either succeed or fail and it is impossible for multiple-clients calling these operations to overwrite or corrupt the stored information.

Asynchronous Operations

Some client libraries support asynchronous requests, where you can ask for a value and then fetch it later. This is useful if you are building an application that builds an UI or has other operations. For example, you might deliberately take advantage of the asynchronous nature like this:

1. (Client) Request retrieval of an item
2. (Client) Builds User Interface (unpopulated)
3. (Server) Retrieves Item; sends to Client
4. (Client) Checks whether requested item could be obtained and displays it

You can request the data to be retrieved, perform the UI setup or formatting, and then retrieve the values. This is particularly useful if some of the values have been stored on disk and therefore take slightly longer to retrieve than those stored in the RAM cache.

Additionally, they can be used during the storage process to store items in the background while other operations are taking place, such as saving a comment while reloading the existing list of comments.

In PHP, the method is called `getDelayed()`, and it accepts one (or more) keys to be retrieved. You can optionally elect to either use a callback function for each document ID/document pair that are returned, or you can use the `fetch()` or `fetchAll()` functions to get the information back.

For example, using the callback method, you could format the returned documents using code similar to the following:

```
$format_recipe = function($key, $value) {
   return ('<li>' . $value['title'] . '</li>');
   };

$ret = $cb->getDelayed(array('recipe1','recipe2'),0,$format_recipe);
```

The callback function is supplied two arguments, the document ID and document of each returned item.

Using `fetch()` you can achieve the same result:

```
$ret = $cb->getDelayed(array('recipe1','recipe2'),0,$format_recipe);

while ($ret = $cb->fetch()) {
echo('<li>' . $ret[value]['title'] . '</li>');
```

In this case we've retrieved each individual document from the database.

Pessimistic Locking

Optimistic locking in Couchbase tries to prevent the updating of a document without the right CAS value. It's optimistic because it doesn't protect against the fact that a client can still update the document by using a straightforward `set()` function to update the document value. Pessimistic locking within Couchbase relies on implementing an explicit locking based mechanism that is ultimately unlocked by updating the value using a CAS compatible function, by explicitly unlocking (using the CAS value), or allowing the lock to optionally time out. With pessimistic locking, it's impossible to update without using a CAS compatible update function.

Usually locking is used as a more explicit method of enforcing the concurrency elements. While a lock is in place, other clients can obtain the value, but they cannot update it without using a CAS compatible update *and* having the right CAS value. This enables you to enforce the use of CAS for updates:

To use the lock, you explicitly perform a get operation with an embedded lock request:

```
$recipe = $cb->getAndLock('recipe1', &$cas);

$recipe = 'new value';

# This will fail, because we are not supplying the CAS value
```

```
$cb->set('recipe1', $recipe);

# This will succeed and then unlock document

$cb->set('recipe1', $recipe, 0, $cas);
```

You can also request a lock with a lock expiry. This locks the document for the specified duration, preventing updates until the lock expiry times out.

The lock can also be explicitly released by using the `unlock()` method:

```
$cb->unlock('recipe1',$cas);
```

There's a default timeout (30 seconds) on the lock to prevent deadlock situations where an item has been locked but never explicitly unlocked.

Deleting Data

There will always be occasions when you want to explicitly delete information from the database, and the `delete()` method will handle these requests. It accepts the document ID, and an optional expiry value. To delete an item immediately, call `delete()`:

```
$cb->delete('message');
```

Once the item has been deleted, further operations that access the document will operate as if the document had never existed, including `get()` and `replace()`.

CHAPTER 5

Storing and Updating Recipes

Now that we've looked at the document design and the core storage and retrieval methods, let's look at some specific code examples using our recipe data.

Creating your document and storing it are straightforward. Depending on your client language you can model the data and use serialization to create the JSON structure.

Initial Storage

Storing your initial document requires creating the document structure (as described above), and then using a `set()` operation to store the data into the database using an appropriate document ID, such as a UUID.

For example:

```
$recipe = array('title' => 'Lasagne', 'subtitle' => 'Traditional italian pasta
dish', 'servings' => 4);
$cb = new Couchbase("127.0.0.1:8091", "recipes", "", "recipes");
$cb->set(uniqid(),$recipe);
```

To get the best results out of the Couchbase when it comes to querying and extracting data using views, JSON should be used as the storage format.

Most languages support some form of serialization of an internal object into an external format. Sometimes this is just for the purposes of external storage (for example on disk), but more complex solutions exist for object relational modeling (ORM), where an internal object is translated into RDBMS based storage.

With Couchbase Server an alternative model is available, with internal objects being serializable into the JSON format, which allows integration with the built-in document model, views and indexing.

Within the PHP client library there are a number of optional connection options that configure how the client library behaves. One of the supported options is enable JSON

serialization which will automatically serialize data objects to and from JSON format as the data is stored in Couchbase Server. You can enable it by setting connection option `OPT_SERIALIZER` to `SERIALIZER_JSON`:

```
...
$cb->setOption(OPT_SERIALIZER, SERIALIZER_JSON);
...
```

The result is that the information will be serialized into, and out of, JSON as you store different objects.

Alternatively, you can manually encode and decode JSON using the `encode_json()` and `decode_json()` functions.

Editing

Editing documents within Couchbase is less about the specifics of writing the queries to update the information, and more about the document loading, manipulation and saving back implied by the document model. You need to edit the entire document, even if you are only updating the value of a single field within the document, or appending data to an existing array. You cannot part-edit a document.

Editing a document therefore follows this basic sequence:

1. Load document from Couchbase using Document ID.
2. Edit the document structure with the new information.
3. Save the updated document back using the document ID (using direct set, locking, or CAS).

For example, to update the title within a recipe, you might use code like this:

```
$cb = new Couchbase("127.0.0.1:8091", "recipes", "", "recipes");
$cb->setOption(OPT_SERIALIZER, SERIALIZER_JSON);

$recipe = $cb->get("00118CCC-3027-11E2-BB0D-B67A7E241592");

$recipe['title'] = 'Lasagne Al Forno';

$ret = $cb->set("00118CCC-3027-11E2-BB0D-B67A7E241592", $recipe);
```

Ideally, for best results, you want to use one of the CAS methods to ensure somebody hasn't updated the document you want to update:

```
$cb = new Couchbase("127.0.0.1:8091", "recipes", "", "recipes");
$cb->setOption(OPT_SERIALIZER, SERIALIZER_JSON);
$cas = '';

$recipe = $cb->get("00118CCC-3027-11E2-BB0D-B67A7E241592", &$cas);
```

```
$recipe['title'] = 'Lasagne Al Forno';

$ret = $cb->set("00118CCC-3027-11E2-BB0D-B67A7E241592", $recipe, 0, $cas);
```

You should check the return value. Handling the situation when the CAS value does not match is more complicated. You will need to determine whether the correct response is to update the information anyway (and just get a new CAS value), or if a more complex merge and update is required. For some as simple as a title, a direct update might be appropriate. If you were updating or expanding the ingredient information, you might want to do a more in depth comparison of the old and the new values.

Loading Recipe

Loading a recipe and displaying it requires knowing the document ID of the recipe that you want to load. We'll look in Chapter 6 at how to perform queries to obtain an appropriate document ID. Until then, getting all the recipe information is as easy as:

```
$cb = new Couchbase("127.0.0.1:8091", "recipes", "", "recipes");
$cb->setOption(OPT_SERIALIZER, SERIALIZER_JSON);

$recipe = $cb->get("00118CCC-3027-11E2-BB0D-B67A7E241592");
```

Now you have the recipe as an internal hash object within PHP. You can format and display that information how you like.

Storing Related Data

Comments are a simple example of the additional information you might store. We'll look later at the methods available for loading information on a unique ID from related documents. If you have used the document structure of the main document to store the related comments, then the update process has two stages; writing the initial information (the comment), and writing the updated recipe data. For example:

```
$cb = new Couchbase("127.0.0.1:8091", "", "", "recipes");
$cb->setOption(OPT_SERIALIZER, SERIALIZER_JSON);

...

my $commentid = uniqid();

$commentret = $cb->set($commentid,$comment);

if ($commentret) {
  $recipe = $cb->get("00118CCC-3027-11E2-BB0D-B67A7E241592");
  array_path($recipe[comments],$commentid);
  $ret = $cb->set("00118CCC-3027-11E2-BB0D-B67A7E241592", $recipe, 0, $cas);
}
```

Now we have successfully added the comment, and then updated the recipe structure to include the ID of the comment, and then updated the recipe.

Loading Related Data

Related records stored alongside the main record data can be loaded either individually, or in bulk, by accessing the list of related record IDs. In our comments example, the `comments` field within the recipe contains a list of all the comment document IDs. Loading all these comments into the system can be achieved by first loading the recipe, then loading the related comment documents:

```
$cb = new Couchbase("127.0.0.1:8091", "", "", "recipes");
$cb->setOption(OPT_SERIALIZER, SERIALIZER_JSON);

$recipe = $cb->get("00118CCC-3027-11E2-BB0D-B67A7E241592");

$comments = $cb->getMulti($recipes['comments']);
```

We've used `getMulti()` to load these in one go, and it's easy now to iterate over the returned array to display the comment information.

Documents Aren't Everything

Although JSON documents enable you to structure and format your information effectively, they are not vital or essential for every type of information that you might want to store into Couchbase Server. You should pick and choose the right format appropriate your application. You may find that certain documents benefit from a simple bytestring or integer format, while others benefit from the JSON structure. Here are some examples where a document may be not be required:

Counters
: If you want to store a simple counter, for example a record of the number of times a recipe has been viewed, then it is much simpler to use a simple integer stored in a document ID and then use the built-in `incr` operation to increment the counter.

Lists or Sequences
: If you are only ever appending a list or structure, it can sometimes be more convenient to use the append or prepend operations to update the data in-situ. For example, if you create a document and always append string and comma ('recipe1,') to the document, you can easily retrieve it and split out the list by splitting on the comma and ignoring the empty last item.

Serialized Native Objects
: For some internal objects that you are using within the structure of your application, you may never need to query or store the data using views, or to manipulate the information except as a native object. Session information, for example, may only

ever be used as an entire object and never queried—just accessed by the session ID each time. Serializing that data out to JSON and back in again each time is resource hungry.

For more information on modeling documents and when to use different formats, check the Couchbase Developer Guide.

CHAPTER 6
Views and Queries

Once you've got your data into Couchbase Server, and you've thought about how you might model and represent the different parts of your information, you want to start pulling that information out again. Selecting the information by knowing the document ID will only get you so far.

Looking back at our recipe data, some frequent questions that might be asked are:

- I've got carrots in my fridge, what can I do with them?
- My bus leaves in 30 minutes, what I can cook in 20 minutes?

Or for the really fussy eaters:

- I've got carrots and 20 minutes, what do I do?

These are not uncommon questions or perspectives on your data, and shouldn't be unfamiliar concepts to any database user. But how do you translate your requirements from the data that we have stored into the list of recipes that match our selection?

The process that handles that is called the view. One or more views are used to process the stored documents and, in the process, create indexes. The view creates a structure that can be used to query and select information from the bucket by making use of the index structures you define.

Creating Views on Your Data

The principles of map/reduce within Couchbase are very simple. In order to query the documents in your database (rather than just access them by their ID), you have to create a view. The view contains the definition for two functions, map and reduce. The map function is required, and it defines the output format and contents of the view. The

reduce function is optional, and is used to summarize, or reduce, the output from the map.

You can choose whether to use the reduce function at the point of querying, which means that a view can serve two purposes, the main querying and the reduced version, and combining view functionality this way helps to reduce the index size and processing requirement for updating it. Common reduce functions include sums and counts, but can be quite complex to produce different reductions and summaries.

To produce the information, the contents of each document are supplied to the map function, which in turn generates a table, which is then processed by the reduce function. The output of both the map and the reduce functions are then stored in an index on disk. You can see this sequence more clearly in the diagram in Figure 6-1.

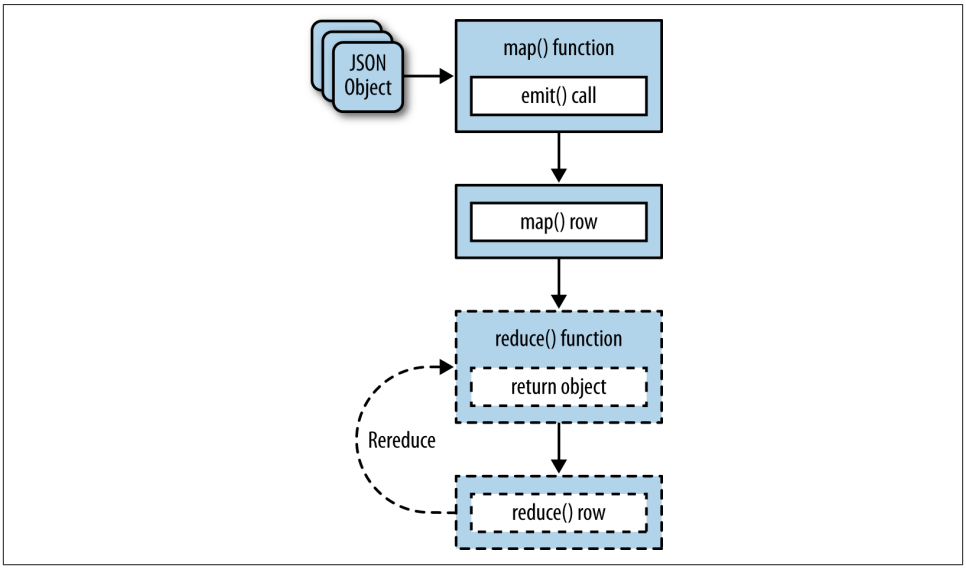


Figure 6-1. Views Workflow

The map and reduce functions are both written in JavaScript, and you can view and edit these functions, and their output and effects, through the Admin Web Console, as shown in Figure 6-2. The admin console allows you to view stored documents, write the map and reduce functions, and get an example of the output generated. You can also select and query the data through the same web console, making it one of the best ways to examine the results, and possible query methods, for your views.

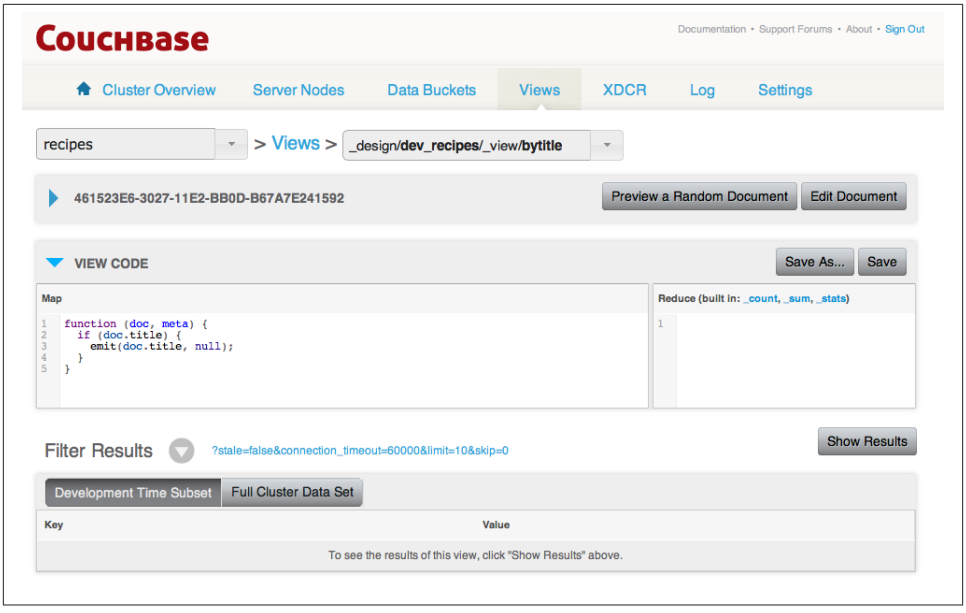


Figure 6-2. Editing Views in the Web Console

The map function outputs two pieces of information, the view key, and the view value, which are unique to the view and independent of the key/value of the document ID and document. The view key and view value are controlled through the use of a call to the function `emit()`. The key is significant because it is the contents of the key that enable you to select information. For example, if you want to search by the recipe title, then you would output the recipe title field in your map function as the key, like this:

```
function (doc, meta)
{
    emit(doc.title, null);
}
```

The above is a very simple map function. The call to `emit()` generates one row of output, and in this case it's output the recipe title.

Maps, Reduce, Views, and Design Documents

The only way to query and search information within Couchbase is by writing an appropriate view. Therefore, for each type of query you make, you need to have created an appropriate view to extract the information.

In fact, views exist within design documents. Each design document can contain multiple views, and each view can contain both the map and optional reduce function. You can define multiple design documents per bucket. For example, within our recipes

bucket we might create multiple design documents, and each design document may contain one or more views, as shown in Figure 6-3.

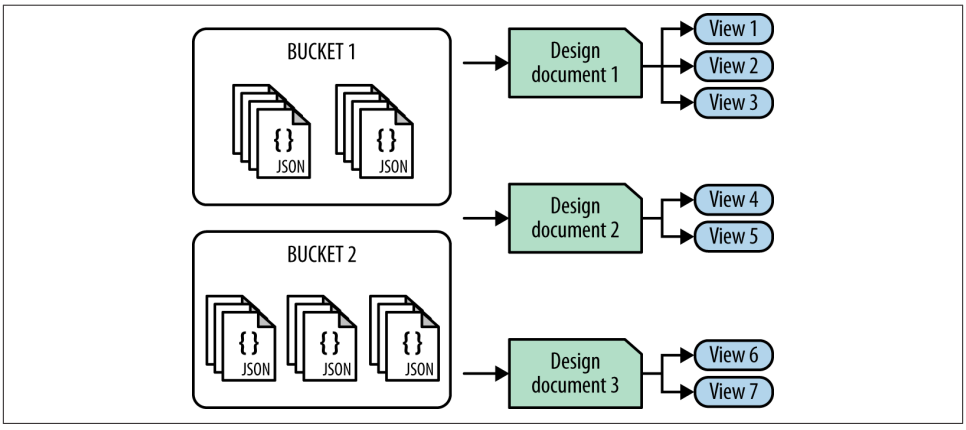


Figure 6-3. Buckets, Design Documents, and Views

The critical distinction is that indexes on disk are created according to design documents. If you have two views in a design document, then both views will be updated at the same time, and the generated information will be stored in the same index. This is important because the update mechanism and rules affect how these are updated.

Both design documents and views must have a name so that they can be identified and output. You can see from Figure 6-4 that I have a number of different views in different design documents, here collected together according to the information they contain. For example, the `dev_bytime` design document contains two views, one by the total cooking time (`bytotaltime`) and one that includes the ingredient and the cooking time (`byingredtime`).

Before we move on and have a look at what a view generates, it's worth noting that views require processing time to create. If you have a lot of documents and write or enable a view for the first time, it has to generate all of the information. During development you don't want this to affect the performance of your production while you are building your view.

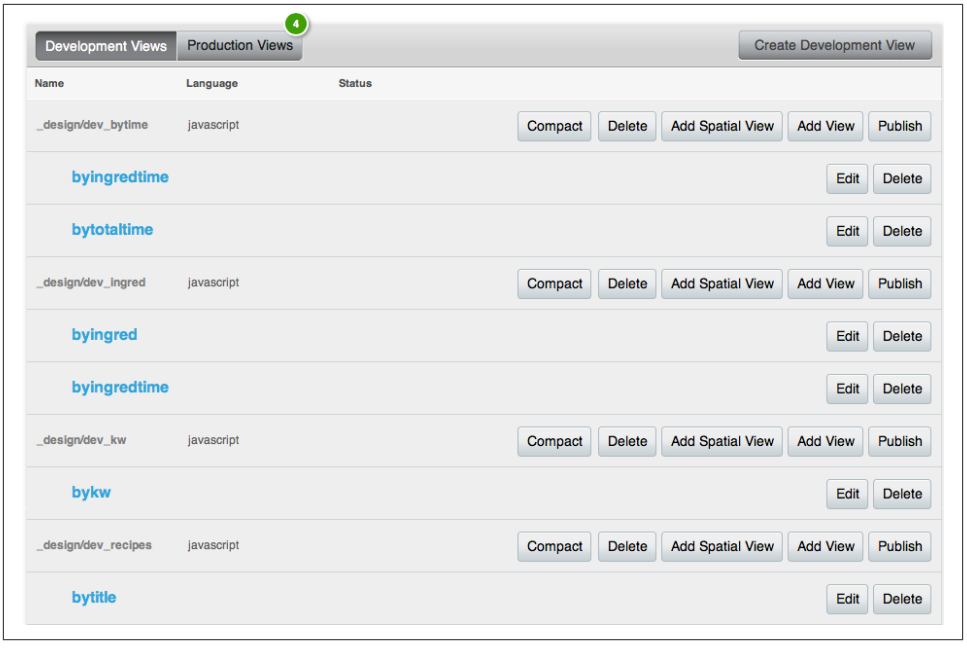


Figure 6-4. Recipe Design Documents and Views

To aid this, Couchbase supports two different classes of design document. Development design documents (prefixed with `dev_`) are classed as in a developmental state. By default, querying these operates on only a small selection of the documents in your bucket (although you can change this). This enables you to write and refine your view to ensure you get the right information, without building a whole new view on the full dataset. This is particularly important during development, since changing the definition of a view immediately invalidates the index, which would normally cause all the documents to require reprocessing.

Once the view is ready to go into production, you can promote a design document to be a production view (no prefix). Production views process all of the documents stored in your bucket, but cannot be modified. This ensures that the index stays up to date according to the view definition and allows for incremental updates to the views as new documents are added and existing documents are updated.

The recommended process for developing and deploying a view is:

1. Create a development view.
2. Update the view until the required content and structure is achieved.
3. Execute the development view over the whole cluster set to create the index data.

4. Promote the view to production state, which will use the index files generated by the previous step.

You can also copy existing production views back into developmental views so that you can edit, update or copy them for new indexes.

View Contents

Now we know how to write a view, what does the output look like? The view output contains three pieces of information for each corresponding call to `emit()`:

Document ID
: Every call to `emit()` includes the document ID which generated the row of information. This allows you to load the full document of data just by doing a `get()` on the document ID.

View key
: This is the first argument from the `emit()` function. You can output any value here, including arrays of values to allow more complex selection and reporting. We'll see how this works in practice shortly. The value of the key is also how you perform querying, the system matches the value you specify in the key when you perform a query.

View value
: This is the second argument from the `emit()` function. It should only be used if you define a reduce function, because the value is also stored in the index along with the key. This can make the size of your indexes very large, and the information is often more easily (and often quickly) available just by performing a `get()` using the supplied document ID.

It's probably best to get the idea by having a look at the raw output. Internally, the view output is just a JSON document. Here's the first 10 rows from our recipe title view as raw JSON:

```
{"total_rows":40,"rows":[
{"id":"40769B5C-2FF2-11E2-86AF-919C63DA47F0","key":"Apple pie","value":null},
{"id":"CFD6F1B8-3027-11E2-BB0D-B67A7E241592","key":"Apricot cheesecake",
"value":null},
{"id":"FCAB10D0-3026-11E2-BB0D-B67A7E241592","key":"Barbecued aubergine",
"value":null},
{"id":"00534ACC-3027-11E2-BB0D-B67A7E241592","key":"Barbecued beefburgers",
"value":null},
{"id":"537879E8-3027-11E2-BB0D-B67A7E241592","key":"Barbecued corn on the cob
slices","value":null},
{"id":"C6AFBEDA-3027-11E2-BB0D-B67A7E241592","key":"Beef in red wine",
"value":null},
{"id":"77543082-3027-11E2-BB0D-B67A7E241592","key":"Blue cheese and tomato
dip","value":null},
```

```
{"id":"96E0AB06-3027-11E2-BB0D-B67A7E241592","key":"Cajun chicken",
"value":null},
{"id":"BB853E90-3027-11E2-BB0D-B67A7E241592","key":"Caribbean cobbler stew",
"value":null},
{"id":"03447F30-3027-11E2-BB0D-B67A7E241592","key":"Chappati or roti",
"value":null}
]
}
```

I've kept the document ID (`id`), key and value for each call to `emit()` on one line here to make the individual rows clearer.

Accessing Views from a Client Library

The view system is actually available through a REST endpoint, but the client libraries handle the query and JSON processing for you to convert the information into a native structure appropriate to your language. In PHP, you make a call to the `view()` method, which takes the design document name, and the view name, and returns the information as an associative array within PHP. For example, to query our Recipes by Title view we might use code like the following to produce a list:

```
<table>
<tr><td>Recipe Title</td></tr>
<?php
$cb = new Couchbase("127.0.0.1:8091", "recipes", "", "recipes");

$result = $cb->view("dev_recipes", "bytitle");
foreach($result["rows"] as $row) {

  echo '<tr><td>',$row['key'],'</td></tr>';
}
?>
</table>
```

Here we connect to the Couchbase Server, access the view, and iterate over the rows of the returned structure to output each recipe title.

Querying and Selection

Querying within views is a function of specifying the keys from the `emit()` function that you want to select from the view and generated index.

Couchbase supports three selection mechanisms:

- Specific key
- One or more keys
- Key range

For example, to query our recipes by title view and get all the recipes with a title that matches "`Lasagne`", Couchbase literally matches the supplied key value to the emitted keys from the view. When the match, the view row is returned. Within PHP, you specify the key to be matched by supplying an associative array with the parameter "`key`" and the corresponding value:

```
$result = $cb->view("recipes", "bytitle", array('key' => 'Apple Pie'));
```

Note that the key matching here is explicit, if I'd specified "`apple pie`" (lowercase), I'd get no responses. The key selection *must* match the emitted key.

Because we match by the key exactly, if there had been multiple recipes with the title "`Apple Pie`", all would have been returned. However, we wont get recipes that simple contain the string "`Apple Pie`." Using `key` gives us an exact match.

You can only query based on what the view generates. If you need to support lowercase queries, then you will need to output the recipe title in the view converted all to lower case, and convert your user's selections to lowercase in the view specification.

Obviously this might be a bit restrictive, so you can also specify more than one value to be selected by supplying a list of keys using the `keys` parameter:

```
$result = $cb->view("dev_recipes", "bytitle", array('keys' => array('Shepherds
pie', 'Mariners pie')));
```

Again, the match is explicit, but we are now matching more than one possible output key value.

Finally, we can get a range of values. All views sort their output according to the UTF-8 value of the key (essentially alphabetically). So we can get all of the recipes between Meat loaf and Mexican tacos using:

```
$result = $cb->view("dev_recipes", "bytitle", array('startkey' => 'Meat
loaf','endkey' => 'Mexican tacos'));
```

Couchbase actually outputs all of the rows where the emitted key is lexically equal to or greater than `startkey`, and stops when the emitted key is lexically greater than `end key`. This means that if we didn't have a meat loaf recipe, we'd still get recipes like "meat-free cottage pie" (which is lexically greater than "meat loaf"), and we'd still stop before the emitted key was "mexican style avocado dip."

We can use this sorting to our advantage. If, for example, we want to obtain all of the recipes that start with "Mediterranean" then we can use the fact that the string "Mediterranean" with the UTF-8 character 0xFFFF appended will always come at the end of words beginning with "Mediterranean" (since 0xFFFF is the largest character).

```
$result = $cb->view("dev_recipes", "bytitle", array('startkey' => 'Mediterra-
nean','endkey' => 'Mediterranean\uFFFF'));
```

The combination of the sorting and the use of special characters is a common theme in querying in Couchbase through views.

Other Options

The key selection parameters are just a small taste of the full parameter set. Views also allow you to page results, reverse the sort order, and other tricks. Check out the documentation for full information, but check the following table for a quick list of the parameters you can include in the array during the call to `view`.

Parameter name	Description
`descending`	Return the documents in descending by key order
`endkey`	Stop returning records when the specified key is reached. Key must be specified as a JSON value.
`endkey_docid`	Stop returning records when the specified document ID is reached
`full_set`	Use the full cluster data set (development views only).
`group`	Group the results using the reduce function to a group or single row
`group_level`	Specify the group level to be used
`inclusive_end`	Specifies whether the specified end key should be included in the result
`key`	Return only documents that match the specified key. Key must be specified as a JSON value.
`keys`	Return only documents that match each of keys specified within the given array. Key must be specified as a JSON value. Sorting is not applied when using this option.
`limit`	Limit the number of the returned documents to the specified number
`on_error`	Sets the response in the event of an error. `stop` will stop returning rows; `continue` will notify you of the error, but continue returning rows from other nodes.
`reduce`	Use the reduction function.
`skip`	Skip this number of records before starting to return the results
`stale`	Allow the results from a stale view to be used. `ok` uses a stale index; `false` forces an index update; `update_after` updates the index after it has been accessed (default)
`startkey`	Return records with a value equal to or greater than the specified key. Key must be specified as a JSON value.
`startkey_docid`	Return records starting with the specified document ID

Of these, probably the most important to understand is how you can control when indexes are updated. For more information, see "Index Updates" (page 58).

Dealing with Different Document Formats

Because the function is JavaScript, we can perform all sorts of checks and manipulation on the information before we call `emit()` to output a row of information for the view.

We'll see some examples of this shortly, but keep in mind that we can call `emit()` multiple times, and we could pick and choose different fields and values.

For example, if the version of a recipe record had changed over time, we might want to output the correct field according to the input version. For example, if we had initially created the recipe with just the total cooking time, and then expanded the single `totalcooktime` field in separate fields for preparation and cooking we might write a view like this:

```
function (doc, meta)
{
  if (doc.preptime && doc.cooktime)
  {
    emit(parseInt(doc.preptime, 10) + parseInt(doc.cooktime, 10), null);
  }
  else
  {
    emit(parseInt(doc.totalcooktime, 10), null);
  }
}
```

We now have a view and querying mechanism that can still operate on the same query requests (recipes that take less than 20 minutes), even though the underlying document data structure might have changed.

View Values and Reduction

The view value (from the call to `emit()`) should only be used if you want to reduce the information in some way to provide a summary. The value can be null, as in the above examples, if we don't want to use the value in any way. Reduce functions are entirely optional, but they are used by default if they are defined. But, at the point of querying, you can choose whether to use the reduce function or not.

For example, you might have written a view that outputs the list of recipe keywords, and a reduce function that counts them. At the point of querying, I can choose whether I want to use the reduce function or not. This allows for a view to have a dual purpose, and be used for both querying on the data, and producing summaries of the data. We'll take a closer look at some reduce functions later in this chapter.

Index Updates

Because of the distributed nature of Couchbase Server, the processing of views and indexes is subject to eventual consistency. Specifically, the indexes are updated according to specific criteria and may be logically out of synchronization with the documents that have been stored.

The reason for this is the way in which documents are first eligible for inclusion in the index, and then how the index is ultimately updated:

- The main reads and writes (`get()` and `set()`) to the server always go through the RAM caching layer. A document only becomes eligible for inclusion in any index when once the document has been persisted down to disk. Until this persistence of the document to disk occurs, the document (or an update to a document) will not appear in any index.

  You can see this sequence clearly in Figure 6-5.

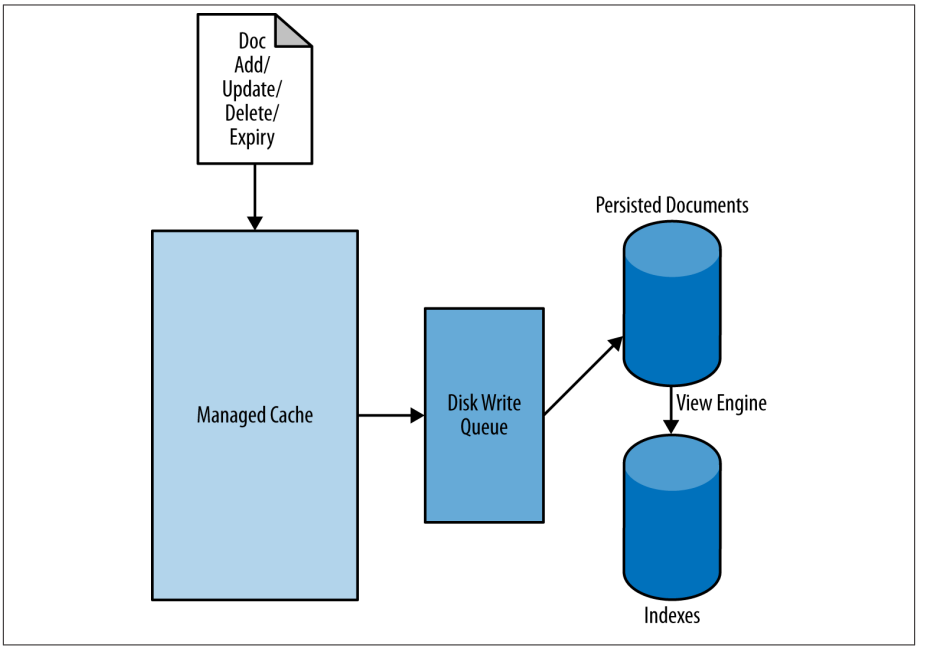


Figure 6-5. Editing Views in the Web Console

  If the change to the document has not yet been written to disk, the contents will never make it into the index.

- An index can be automatically updated by the server. You can configure a number of documents, or time period, or both, when the index is updated.
- At the point of querying, you can specify whether the index should be updated before the query is executed, after the query is executed, or not updated at all (stale indexes).

- As with updates and new document additions, when a document is deleted from the database, it is only removed from an index once the removal has been applied to the on-disk version of the document.
- If the document has an expiry (TTL), then the document is only removed from the index once the document has expired, either through the operation of the background expiry pager or because the deleted document has been accessed in some way.

This eventual consistency for the index requires some careful management in your applications. To aid your application, you can use the persistence and observe functions to determine whether a document update had been stored onto disk. For example, to ensure that a document has persisted to disk during a `set()` operation within PHP, which accepts a fifth argument, `persist_to`. This argument specifies the number of servers on which the document should have been persisted to disk before the operation returns:

```
$cb_obj->set("spoon", "Hello World!", 10, NULL, 1);
```

The use of 1 in the above example will ensure that the `set()` operation does not return until the document has been persisted to disk. Once persisted, the document should be in any view that processes this document.

Stale Indexes and Updates

Once documents have been persisted (and are therefore eligible for inclusion in a view), you then need to determine whether the view index should be updated. You can control this by requesting an update for all eligible documents during the query using the `stale` parameter. This accepts one of three values:

`update_after`
: The default, this updates the index for all eligible documents after processing the current query. This means that the next query will get an updated version of the view.

`ok`
: Execute the query using the current version of the index. This returns the matching documents using the stale, and therefore not updated version of the index.

`false`
: Force an update of the index for all eligible documents, then execute the query and return the results. This forces an update, and may therefore take more time to return the query results, since the updated documents must be processed by the map/reduce functions before the results are returned.

Using a combination of the persist/observe and `stale` parameter enables your application to be very specific about when the information is updated. For example, you might

be happy to use a "stale" index for recipe data (which is updated comparatively infrequently), but use persistence and `stale=false` when storing a recipe comment so that the user comment appears immediately.

In a distributed and concurrent system, there are of course additional complexities. It's not possible, for example, to only update the index with certain documents. All document changes that have been persisted to disk since the last update will be processed. You will need to modify and develop your application with these principles in mind to optimize the experience according to the number of users and concurrency required.

Searching and Querying Examples

Hopefully it should be clear by now that querying data in Couchbase is a function of creating a suitable `map()` to define the information that can be queried, and then specifying a suitable key selection (single, multi, or range) to get the information you want. Let's take a look at some typical examples so that you can see how this works in practice.

We've already seen some basic searches by recipe title. Let's look at some the solutions to the recipe requirements introduced at the start of this chapter.

Searching By Ingredient

Within our recipe data, ingredients are stored as an associative array within an array of possible ingredients. To allow us to query on ingredients, we need to call the `emit()` function for each ingredient within each recipe. Within JavaScript we can do that by iterating over the ingredient list, and calling `emit()` with the ingredient name from the ingredients' array, like this:

```
function(doc, meta)
{
  if (doc.ingredients)
  {
    for (i=0; i < doc.ingredients.length; i++)
    {
      if (doc.ingredients[i].ingredient) {
        emit(doc.ingredients[i].ingredient, null);
      }
    }
  }
}
```

This generates multiple view rows from each recipe, one for each ingredient, giving a JSON structure like this:

```
{"total_rows":382,"rows":[
{"id":"E785D4EC-3026-11E2-BB0D-B67A7E241592","key":"anchovy fillets",
"value":null},
{"id":"FCAB10D0-3026-11E2-BB0D-B67A7E241592","key":"aubergines","value":null},
{"id":"3E9E0358-3027-11E2-BB0D-B67A7E241592","key":"avocado","value":null},
{"id":"C6AFBEDA-3027-11E2-BB0D-B67A7E241592","key":"bay leaf","value":null},
{"id":"6466C8EA-3027-11E2-BB0D-B67A7E241592","key":"bay leaves","value":null},
{"id":"AAF9D658-3027-11E2-BB0D-B67A7E241592","key":"bay leaves","value":null},
{"id":"6466C8EA-3027-11E2-BB0D-B67A7E241592","key":"beef braising steak",
"value":null},
{"id":"C6AFBEDA-3027-11E2-BB0D-B67A7E241592","key":"beef braising steak",
"value":null},
{"id":"6466C8EA-3027-11E2-BB0D-B67A7E241592","key":"beef stock","value":null},
{"id":"99BE8A1E-3027-11E2-BB0D-B67A7E241592","key":"beef stock","value":null}
]
}
```

To select all of the recipes that contain carrots (or first use case question):

```
$result = $cb->view("dev_ingred", "byingred", array('key' => 'carrots'));
```

Note that the information returned will only include the ingredients that were originally selected in the key. You can use `get()` to obtain the document value. A full script might look like this:

```
<table>
<tr><td>Recipe Title</td></tr>
<?php
$cb = new Couchbase("192.168.0.99:8091", "recipes", "", "recipes");

$result = $cb->view("ingred", "byingred", array('key' => 'carrots'));
foreach($result["rows"] as $row) {
                $recipe        =        json_decode($cb->get($row['id']),
true);

  echo '<tr><td>',$recipe["title"],'</td></tr>';
}
?>
</table>
```

Yay, a full list of recipes that contain carrots!

Searching by Recipe Time

To create a search that looks at a different field, or combination, we need to create another different view, this time to output the cooking time:

```
function (doc, meta) {
  if (doc.title) {
    emit(parseInt(doc.totaltime, 10), null);
  }
}
```

The `parseInt()` function ensures that we output an integer, rather than a string. To query, we could be explicit and look only for recipes that take exactly 10 minutes to cook:

```
$result = $cb->view("time", "bytime", array('key' => 10));
```

But it would be much more practical to actually look for recipes that could be cooked in 10 minutes or less by using a range query:

```
<table>
<tr><td>Recipe Title</td><td>Time</td></tr>
<?php
$cb = new Couchbase("192.168.0.99:8091", "recipes", "", "recipes");
$cb->setOption(Couchbase::OPT_SERIALIZER, Couchbase::SERIALIZER_JSON);

$result = $cb->view("bytime", "bytime", array('startkey' => 0, 'endkey' => 10));
foreach($result["rows"] as $row) {
  $recipe = json_decode($cb->get($row['id']), true);
  echo '<tr><td>',$recipe["title"],'</td><td>',$recipe['totaltime'],'</td>
</tr>';
}
?>
</table>
```

This works because the view is outputting the individual rows in UTF-8 order, which also happens to be numerical order, so the output will be a list of all the recipes that take from 0 to 10 minutes to prepare, in cooking time order.

Searching by Ingredient and Time

Let's move on to the last question, how to find recipes that both contain carrots and are cookable in 20 minutes. To achieve this, we need to write a view that outputs a compound key, an array, that contains both the ingredient and the total cooking time.

The compound key is easy to define within JavaScript by using square brackets to denote the array, like this:

```
function (doc, meta) {
  emit([parseInt(doc.totaltime,10), doc.title], null);
}
```

Here we're outputting the cooking time first, followed by the recipe title. The key specification must match, that is, it must be an array:

```
$result = $cb->view("bytime", "bytimetitle", array('startkey' => array(0, "La-
sagne"),  'endkey' => array(90, "Lasagne"),));
```

With a compound view key like this you need to think about how the information will be searched to get the values in the compound key the right way round. The `startkey` and `endkey` parameters are merely indicators of when to start, and when to stop, outputting the rows from the view.

This is significant, because the sorting (i.e., the order of the keys) and these parameters will alter the list of the information returned. For example, using the above search, we'll actually get a lot of items we didn't expect that happen to have a 0 at the start, because `startkey` only specifies when to start outputting the information. The first view key from our recipe database that is lexically greater than `[0, "Lasagne"]` is in fact `[0,"Mixed iceberg salad with creamy basil dressing"]`. That's not even Lasagne!

In this case, the "`Lasagne`" is a fixed query value, but the time is a variable (or range) query value, and therefore we should reverse the view:

```
function (doc, meta) {
  emit([doc.title, parseInt(doc.totaltime,10)], null);
}
```

And the view query:

```
$result = $cb->view("bytime", "bytimetitle", array('startkey' => array("La-
sagne", 0), 'endkey' => array("Lasagne", 90),));
```

Now we will correctly get all the recipes that have "`Lasagne`" as the title, but which can be cooked within 90 minutes, because the sorting will list `["Lasagne", 0]` first, and `["Lasagne", 90]` lexically in order.

We can apply that to our ingredients, first by iterating over the list of ingredients, and then adding the cooking time:

```
function (doc, meta) {
 for (i=0; i < doc.ingredients.length; i++)
 {
   if (doc.ingredients[i].ingredient != null) {
     emit([doc.ingredients[i].ingredient, parseInt(doc.totaltime, 10)], null);
   }
 }
}
```

We can look for carrot recipes that can be cooked in less than 20 minutes using this script:

```
<table>
<tr><td>Recipe Title</td><td>Time</td></tr>
<?php
$cb = new Couchbase("192.168.0.99:8091", "recipes", "", "recipes");
$cb->setOption(Couchbase::OPT_SERIALIZER, Couchbase::SERIALIZER_JSON);

$result = $cb->view("bytime", "byingredtime", array('startkey' => array("car-
rots", 0), 'endkey' => array("carrots", 20)));
foreach($result["rows"] as $row) {
  $recipe = json_decode($cb->get($row['id']), true); echo
  '<tr><td>',$recipe["title"],'</td><td>',$recipe['totaltime'],'</td></tr>';
}
?>
</table>
```

And that gives us a final list of recipes we can cook before going out:

```
Recipe Title Time
Mango and carrot smoothie 5
Crispy crumbed cauliflower and carrots 8
Carrots with lemon and sherry 12
Carrots in herby mayonnaise 13
Caramelised carrot sticks 13
Mixed steamed vegetables 13
Raisin coleslaw 15
Steamed citrus vegetables 15
Warm lemony tricolour julienne vegetables 15
Scottish style vegetables 15
Cheese and apple coleslaw 15
Onion vegetable medley 15
Chicken, leek and sun-dried tomato soup 16
Orange glazed carrots - microwave 16
Corned beef bubble and squeak 16
Orange glazed carrots - microwave 16
Chinese satay style noodles 17
Spiced orange and passion cake 19
Carrot fettucine 19
Sweet and sour vegetable potato filler 20
Tofu noodles 20
Carrots and parsnips with orange and coriander 20
Platter of mixed hors d'oeuvres 20
Eastern mushroom and vegetable couscous 20
Spicy chilli prawns 20
Platter of mixed hors d'oeuvre 20
```

Success!

Writing views is not as complicated as it first sounds, but, you do need to ensure that you have built the correct view before you can start to query, and sometimes that will require careful planning. For more examples of different views and querying, check the Couchbase documentation which includes a number of detailed (and more complex) examples.

Reductions

The reduce functions are designed to summarize the information that you are creating. All of the built-in functions are designed to work with one or more values, either as a single value, or an array of values, and are all optimized for data operation within the Erlang layer of the View system. Custom reduce functions operate within the JavaScript layer.

The Built-in _count Function

The `_count` function provides a simple count of the input rows from the `map()` function, using the keys and group level to provide a count of the correlated items. The values generated during the `map()` stage are ignored.

Counting is a great way of providing top ten lists and other output where you want the most popular or regularly used recipes, keywords or other elements to be reported. For example, we can produce a list of keywords and the number of recipes that contain them using a `map()` function like this:

```
function (doc, meta) {
  if (doc.keywords) {
    for (kw in doc.keywords) {
      emit(kw,null);
    }
  }
}
```

Specifying a built-in function of `_count` and accessing the view, outputs a useful list:

```
{
   "rows" : [
      {
         "key" : "convenience@add bread for complete meal",
         "value" : 242
      },
      {
         "key" : "convenience@add jacket potato for a complete meal",
         "value" : 116
      },
      {
         "key" : "convenience@add pasta for a complete meal",
         "value" : 48
      },
      {
         "key" : "convenience@add rice for a complete meal",
         "value" : 135
      },
      {
         "key" : "convenience@serve with salad for complete meal",
         "value" : 217
      },
      {
         "key" : "cook method.hob, oven, grill@grill",
         "value" : 141
      },
      {
         "key" : "cook method.hob, oven, grill@grill / oven",
         "value" : 9
      },
      {
```

```
            "key" : "cook method.hob, oven, grill@hob",
            "value" : 1058
        },
        {
            "key" : "cook method.hob, oven, grill@hob / grill",
            "value" : 101
        },
        {
            "key" : "cook method.hob, oven, grill@hob / grill / oven",
            "value" : 8
        }
    ]
}
```

The _sum Function

The sum function adds one or more values from the output of the `map()` function together. We could use as a better way to output the total cooking time for a given recipe, rather than having to manually update the `totalcooktime` field in our recipe documents. To do this, first we have to output the individual timing fields from the document in the `map()` function:

```
function (doc, meta) {
  emit(doc.title, parseInt(doc.preptime, 10));
  emit(doc.title, parseInt(doc.cooktime, 10));
}
```

Because values in JSON can sometimes be stored as strings as well as numerical values, when writing your map, using `parseInt()` or `parseFloat()` to cast your data into the right type is vital. Otherwise, the `reduce()` function will raise an error and your view will be empty.

If we group the output, then the total cooking time for each recipe is automatically generated for us:

```
{
   "rows" : [
      {
         "key" : "",
         "value" : 0
      },
      {
         "key" : "'Goat' curry",
         "value" : 65
      },
      {
         "key" : "3-tier salmon, spinach and avocado terrine",
         "value" : 38
      },
```

```
        {
           "key" : "Aberffraw cake",
           "value" : 27
        },
        {
           "key" : "Aduki and orange casserole - microwave",
           "value" : 47
        },
        {
           "key" : "Aioli - garlic mayonnaise",
           "value" : 40
        },
        {
           "key" : "Alabama peanut chicken",
           "value" : 19
        },
        {
           "key" : "Alebele (Sweet spiced pancakes)",
           "value" : 35
        },
        {
           "key" : "Allspice rice",
           "value" : 17
        },
        {
           "key" : "Almond and cauliflower soup",
           "value" : 58
        }
    ]
}
```

A simple but effective way of determining the total time required to prepare and cook a recipe.

The Built-in _stats Function

The `_stats` records a series of statistics from the mapped values. For example, if you want to get the minimum and maximum times for cooking recipes with different ingredients, you can output each ingredient and the total cooking time in your map:

```
function (doc, meta) {
  if (doc.ingredients) {
    for (i=0; i < doc.ingredients.length; i++)
    {
      if (doc.ingredients[i] != null)
      {
        emit(doc.ingredients[i].ingredient, parseInt(doc.totaltime, 10));
      }
    }
  }
}
```

Using `_stats` as the reduce function now provides us with minimum and maximum cooking times for each ingredient, as well as the total time, a count of the number of times the ingredient is used, and the sum of the squares:

```
{
   "rows" : [
      {
         "value" : {
            "count" : 2,
            "min" : 45,
            "sumsqr" : 4050,
            "max" : 45,
            "sum" : 90
         },
         "key" : "acorn squash"
      },
      {
         "value" : {
            "count" : 12,
            "min" : 10,
            "sumsqr" : 126270,
            "max" : 255,
            "sum" : 886
         },
         "key" : "almond essence"
      }
   ]
}
```

Who would have thought almond essence could create recipes that require three hours to prepare!

Reductions are themselves a big topic, and you should check the official documentation for more information on custom reduce functions, grouping, and query selection when performing reductions. They can create some powerful combinations.

Document Metadata

During view processing, metadata about individual documents is exposed through a separate JSON object, `meta`, that can be optionally used as the second argument to the `map()`. This metadata can be used to further identify and qualify the document being processed.

The `meta` structure contains the following fields and associated information:

`id`
: The ID or key of the stored data object. This is the same as the key used when writing the object to the Couchbase database.

`rev`
: An internal revision ID used internally to track the current revision of the information. The information contained within this field is not consistent or trackable and should not be used in client applications.

`type`
: The type of the data that has been stored. A valid JSON document will have the type `json`. Documents identified as binary data will have the type `base64`.

`flags`
: The numerical value of the flags set when the data was stored. The availability and value of the flags is dependent on the client library you are using to store your data. Internally the flags are stored as a 32-bit integer.

`expiration`
: The expiration value for the stored object. Expiration times are expressed in seconds. If the value is less than 30 days (30*24*60*60), the expiration is the number of seconds until the value expires. A number higher than this indicates an absolute expiration based on the number of seconds since the epoch.

These additional fields are only exposed when processing the documents within the view server. These fields are not returned when you access the object through the Memcached/Couchbase protocol as part of the document.

CHAPTER 7

Next Steps

Hopefully this book has given you an introduction into the main points and requirements when developing applications with Couchbase, but it is not an exhaustive guide to all of the functionality or all of the different use cases and scenarios. In this chapter, we'll look at some pointers for more information on how to get the best out of Couchbase by using the documentation and guides.

Couchbase Server Resources

The main Couchbase Server guide should tell you everything you need to know about deploying a cluster, and then managing and monitoring that cluster for your application.

- Chapter 2 contains everything you need to install and configure your new cluster. See this page.
- Guides on administering and managing your Couchbase Server cluster are located at this page.
- Administration tools are also very helpful, including the Web Console, Command-line Tools, and REST API.
- Writing Views and creating indexes within Couchbase Server is a large topic and there are a range of background mechanics, examples, and sample querying solutions, as you can see at this page.

For all other documentation and examples, go to the main documentation page.

Couchbase Developer Resources

Successful application development within Couchbase Server relies on two different aspects: the core methodologies (such as structuring your data, and modeling your

documents) and the basic processes of storing, retrieving and updating your data, and the language-specific client library that supports that functionality.

The Couchbase Developer Guide provides the background information about how applications should operate with Couchbase Server. Here you'll find information on getting the best performance, handling errors, debugging, and understanding the link between the Couchbase Server and the client library used to interface to it.

For specific help on each of the different client libraries, you should start with the appropriate client library page. This will in turn provide links to the client library, documentation, and example applications:

- Java
- .NET
- PHP
- Ruby
- C
- Python

About the Author

A professional writer for over 15 years, Martin "MC" Brown is the author and contributor to over 26 books covering an array of topics, including programming, system management, and web technologies. His expertise spans a myriad of development languages and platforms—Perl, Python, Java, JavaScript, Basic, Pascal, Modula-2, C, C++, Rebol, Gawk, Shellscript, Windows, Solaris, Linux, BeOS, Microsoft WP, Mac OS, and more. The combination has resulted in expertise in web programming, systems management and integration, and XML and DocBook technologies for writing and publishing documentation. He is a former LAMP Technologies Editor for *LinuxWorld* magazine and is a regular contributor to ServerWatch.com, LinuxPlanet, *ComputerWorld*, and IBM developerWorks. As a Subject Matter Expert for Microsoft, he provided technical input to their Windows Server and certification teams.

He draws on a rich and varied background as a founder member of a leading UK ISP, systems manager and IT consultant for an advertising agency and Internet solutions group, technical specialist for an intercontinental ISP network, and database designer and programmer—and as a self-confessed compulsive consumer of computing hardware and software. In his pre-writing life he spent more than 10 years designing and managing mixed platform environments. As a result he has developed the rare talent of being able to convey the benefits and intricacies of his subject with equal measures of enthusiasm, professionalism, in-depth knowledge, and insight. He is a past technical writer, building both the documentation system and writing content for MySQL and the MySQL groups within Sun and then Oracle. MC is currently the VP of Technical Publications and Education for Couchbase and is responsible for all published documentation, training programs and content, and the Couchbase Community website.

Colophon

The animal on the cover of *Developing with Couchbase Server* is the African softshell turtle (*Trionyx triunguis*).

The cover image is from a loose plate of Hungarian origin, source unknown. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.