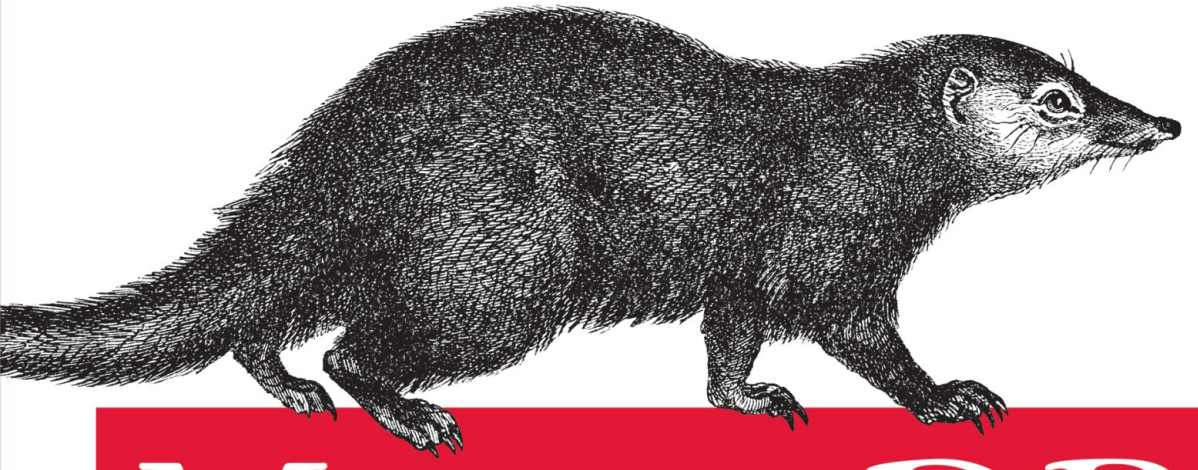
Patterns and Processes for the Popular Document-Oriented Database



MongoDB & Python

O'REILLY®

Niall O'Higgins

MongoDB and Python

Learn how to leverage MongoDB with your Python applications, using the hands-on recipes in this book. You get complete code samples for tasks such as making fast geo queries for location-based apps, efficiently indexing your user documents for social-graph lookups, and many other scenarios.

This guide explains the basics of the document-oriented database and shows you how to set up a Python environment with it. Learn how to read and write to MongoDB, apply idiomatic MongoDB and Python patterns, and use the database with several popular Python web frameworks. You'll discover how to model your data, write effective queries, and avoid concurrency problems such as race conditions and deadlocks.

The recipes will help you:

- Read, write, count, and sort documents in a MongoDB collection
- Learn how to use the rich MongoDB query language
- Maintain data integrity in replicated/distributed MongoDB environments
- Use embedding to efficiently model your data without joins
- Code defensively to avoid KeyErrors and other bugs
- Apply atomic operations to update game scores, billing systems, and more with the fast accounting pattern
- Use MongoDB with the Pylons 1.x, Django, and Pyramid web frameworks

Strata
Making Data Work

Strata is the emerging ecosystem of people, tools, and technologies that turn big data into smart decisions. Find information and resources at oreilly.com/data.

Twitter: @oreillymedia
facebook.com/oreilly

US $19.99 CAN $20.99
ISBN: 978-1-449-31037-0

O'REILLY®
oreilly.com

MongoDB et Python

Niall O'Higgins

O'REILLY®
Pékin Cambridge Farnham Cologne Sébastopol Tokyo

MongoDB et Python par
Niall O'Higgins

Imprimé aux États-Unis d'Amérique.

Publié par O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sébastopol, CA 95472.

Les livres O'Reilly peuvent être achetés à des fins éducatives, commerciales ou promotionnelles. Des éditions en ligne sont également disponibles pour la plupart des titres (http://my.safaribooksonline.com). Pour plus d'informations, contactez notre service des ventes corporatives/institutionnelles : (800) 998-9938 ou corporate@oreilly.com.

Editeurs : Mike Loukides et Shawn Wallace
Monteuse de production : Jasmine Perez
Correcteur : O'Reilly Production Services

Créatrice de la couverture : Karen Montgomery
Architecte d'intérieur : David Futato
Illustrateur : Robert Romano

ISBN : 978-1-449-31037-0

[LSI]

1315837615

Table des matières

Préface. v

1. Mise en route 1
   - Introduction 1
   - Trouver de la documentation de référence 2
   - Installation de MongoDB 3
   - Exécuter MongoDB 5
   - Configuration d'un environnement Python avec MongoDB 6

2. Lecture et écriture sur MongoDB avec Python 9
   - Se connecter à MongoDB avec Python Obtenir dix
   - un handle de base de données 11
   - Insérer un document dans une collection Écrire 12
   - dans une collection de manière sûre et synchrone Garantir 13
   - les écritures sur plusieurs nœuds de base de données 14
   - Introduction au langage de requête MongoDB Lire, 15
   - compter et trier des documents dans une collection Mettre à jour des 15
   - documents dans une collection Supprimer des 18
   - documents à partir d'une collection Opérateurs de 20
   - requête MongoDB Modificateurs de 21
   - mise à jour MongoDB 22

3. Modèles MongoDB et Python courants. 23
   - Un modèle unique orienté document : l'intégration 23
   - Recherches rapides : utilisation d'index avec MongoDB 29
   - Applications basées sur la localisation avec MongoDB : indexation géospatiale 33
   - Codez de manière défensive pour éviter les erreurs de clé et autres bugs 37
   - Mise à jour ou insertion : upserts dans MongoDB 39
   - Lecture-écriture-modification atomique : findAndModify de MongoDB 40
   - Modèle de comptabilité rapide 41

4. MongoDB avec les frameworks Web. 45
Pylons 1.x et MongoDB 45
Pyramid et MongoDB 49
Django et MongoDB 51
vont plus loin 53

Préface

Je crée des applications de production basées sur des bases de données depuis environ 10 ans. J'ai travaillé avec la plupart des bases de données relationnelles habituelles (MSSQL Server, MySQL, PostgreSQL) et avec des bases de données non relationnelles très intéressantes (Graphd/MQL de Freebase.com, Berkeley DB, MongoDB). MongoDB est à ce stade le système avec lequel j'aime le plus travailler et que je choisis pour la plupart des projets. Il se situe quelque part à la croisée des chemins entre la performance et le pragmatisme d'un système relationnel et la flexibilité et l'expressivité d'une base de données du web sémantique. Cela a joué un rôle central dans ma réussite dans la construction de systèmes assez complexes en peu de temps.

J'espère qu'après avoir lu ce livre, vous constaterez que MongoDB est une base de données agréable avec laquelle travailler et qui ne gêne pas entre vous et l'application que vous souhaitez créer.

Conventions utilisées dans ce livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Indique les nouveaux termes, URL, adresses e-mail, noms de fichiers et extensions de fichiers.

Largeur constante

Utilisé pour les listes de programmes, ainsi que dans les paragraphes pour faire référence à des éléments de programme tels que les noms de variables ou de fonctions, les bases de données, les types de données, les variables d'environnement, les instructions et les mots-clés.

Gras à largeur constante

Affiche les commandes ou tout autre texte qui doit être saisi littéralement par l'utilisateur.

Italique à largeur constante

Affiche le texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.

Cette icône signifie un conseil, une suggestion ou une note générale.

Cette icône indique un avertissement ou une mise en garde.

Utilisation d'exemples de code

Ce livre est là pour vous aider à accomplir votre travail. En général, vous pouvez utiliser le code de ce livre dans vos programmes et votre documentation. Vous n'avez pas besoin de nous contacter pour obtenir une autorisation, sauf si vous reproduisez une partie importante du code. Par exemple, écrire un programme utilisant plusieurs morceaux de code de ce livre ne nécessite aucune autorisation. La vente ou la distribution d'un CD-ROM d'exemples tirés des livres d'O'Reilly nécessite une autorisation. Répondre à une question en citant ce livre et en citant un exemple de code ne nécessite pas d'autorisation. L'incorporation d'une quantité importante d'exemples de code de ce livre dans la documentation de votre produit nécessite une autorisation.

Nous apprécions, mais nous ne demandons pas d'attribution. Une attribution comprend généralement le titre, l'auteur, l'éditeur et l'ISBN. Par exemple : « MongoDB et Python par Niall O'Higgins.

Si vous pensez que votre utilisation d'exemples de code ne relève pas d'un usage équitable ou de l'autorisation donnée ci-dessus, n'hésitez pas à nous contacter à permissions@oreilly.com.

Livres Safari® en ligne

Safari Books Online est une bibliothèque numérique à la demande qui vous permet de rechercher facilement plus de 7 500 ouvrages et vidéos de référence technologiques et créatifs pour trouver rapidement les réponses dont vous avez besoin.

Avec un abonnement, vous pouvez lire n'importe quelle page et regarder n'importe quelle vidéo de notre bibliothèque en ligne. Lisez des livres sur votre téléphone portable et vos appareils mobiles. Accédez aux nouveaux titres avant qu'ils ne soient disponibles pour impression, bénéficiez d'un accès exclusif aux manuscrits en développement et publiez des commentaires pour les auteurs. Copiez et collez des exemples de code, organisez vos favoris, téléchargez des chapitres, marquez des sections clés, créez des notes, imprimez des pages et bénéficiez de nombreuses autres fonctionnalités permettant de gagner du temps.

O'Reilly Media a téléchargé ce livre sur le service Safari Books Online. Pour avoir un accès numérique complet à ce livre et à d'autres sur des sujets similaires publiés par O'Reilly et d'autres éditeurs, inscrivez-vous gratuitement sur http://my.safaribooksonline.com.

Comment nous contacter

Veuillez adresser vos commentaires et questions concernant ce livre à l'éditeur : O'Reilly

Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (aux États-Unis ou au Canada)
707-829-0515 (international ou local)
707-829-0104 (fax)

Nous avons une page Web pour ce livre, où nous répertorions les errata, des exemples et toute information supplémentaire. Vous pouvez accéder à cette page à l'adresse suivante :

http://www.oreilly.com/catalog/0636920021513

Pour commenter ou poser des questions techniques sur ce livre, envoyez un e-mail

à : bookquestions@oreilly.com

Pour plus d'informations sur nos livres, cours, conférences et actualités, consultez notre site Web à l'adresse http://www.oreilly.com.

Retrouvez-nous sur Facebook : http://facebook.com/

oreilly Suivez-nous sur Twitter : http://twitter.com/oreillymedia

Regardez-nous sur YouTube : http://www.youtube.com/oreillymedia

Remerciements

Je voudrais remercier Ariel Backenroth, Aseem Mohanty et Eugene Ciurana pour leurs commentaires détaillés sur la première ébauche de ce livre. Je voudrais également remercier l'équipe d'O'Reilly pour avoir pris un grand plaisir à écrire ce livre. Bien sûr, merci à toutes les personnes de 10gen sans qui MongoDB n'existerait pas et ce livre n'aurait pas été possible.

CHAPITRE 1

Commencer

Introduction

Lancé pour la première fois en 2009, MongoDB est relativement nouveau sur la scène des bases de données par rapport aux géants contemporains comme Oracle, dont les premières versions remontent aux années 1970. En tant que base de données orientée document généralement regroupée dans la catégorie NoSQL, elle se distingue parmi les magasins de valeurs clés distribués, les clones d'Amazon Dynamo et les réimplémentations de Google BigTable. En mettant l'accent sur une prise en charge riche des opérateurs et un traitement des transactions en ligne (OLTP) hautes performances, MongoDB est à bien des égards plus proche de MySQL que des bases de données orientées par lots comme HBase.

Les principales différences entre l'approche orientée document de MongoDB et une base de données relationnelle traditionnelle sont :

1. MongoDB ne prend pas en charge les jointures.
2. MongoDB ne prend pas en charge les transactions. Il prend en charge l'atome opérations cependant.
3. Les schémas MongoDB sont flexibles. Tous les documents d'une collection ne doivent pas nécessairement respecter le même schéma.

1 et 2 sont le résultat direct des énormes difficultés rencontrées pour faire évoluer ces fonctionnalités sur un grand système distribué tout en maintenant des performances acceptables. Ce sont des compromis effectués afin de permettre une évolutivité horizontale. Bien que MongoDB manque de jointures, il introduit certaines fonctionnalités alternatives, par exemple l'intégration, qui peuvent être utilisées pour résoudre bon nombre des mêmes problèmes de modélisation de données que les jointures. Bien sûr, même si l'intégration ne fonctionne pas vraiment, vous pouvez toujours effectuer votre jointure dans le code de l'application, en effectuant plusieurs requêtes.

Le manque de transactions peut parfois être pénible, mais heureusement, MongoDB prend en charge un ensemble assez décent d'opérations atomiques. Des opérateurs d'incrémentation et de décrémentation atomiques de base au plus riche « findAndModify », qui est essentiellement un opérateur atomique de lecture-modification-écriture.

Il s'avère qu'un schéma flexible peut être très bénéfique, surtout lorsque vous prévoyez d'itérer rapidement. Même si la conception initiale du schéma, telle qu'elle est utilisée dans le modèle relationnel, a sa place, son coût de maintenance est souvent élevé. Gérer les mises à jour de schémas dans le monde relationnel est bien sûr faisable, mais cela a un prix.

Dans MongoDB, vous pouvez ajouter de nouvelles propriétés à tout moment, de manière dynamique, sans avoir à vous soucier des instructions ALTER TABLE dont l'exécution peut prendre des heures et des scripts de migration de données compliqués. Cependant, cette approche comporte ses propres compromis. Par exemple, l'application du type doit être soigneusement gérée par le code de l'application. La gestion des versions de documents personnalisée peut être souhaitable pour éviter de gros blocs conditionnels pour gérer des documents hétérogènes dans la même collection.

La nature dynamique de MongoDB se prête tout naturellement au travail avec un langage dynamique tel que Python. Les compromis entre un langage typé dynamiquement tel que Python et un langage typé statiquement tel que Java reflètent à bien des égards les compromis entre le modèle flexible et orienté document de MongoDB et la définition de schéma initiale et typée statiquement des bases de données SQL.

Python vous permet d'exprimer des documents et des requêtes MongoDB de manière native, grâce à l'utilisation de fonctionnalités de langage existantes telles que des dictionnaires et des listes imbriqués. Si vous avez travaillé avec JSON en Python, vous serez immédiatement à l'aise avec les documents et requêtes MongoDB.

Pour ces raisons, MongoDB et Python constituent une combinaison puissante pour le développement rapide et itératif d'applications backend évolutives horizontalement. Pour la grande majorité des applications Web et mobiles modernes, nous pensons que MongoDB est probablement mieux adapté que la technologie SGBDR.

Trouver de la documentation de référence

MongoDB, Python, le pilote PyMongo de 10gen et chacun des frameworks Web mentionnés dans ce livre disposent tous d'une bonne documentation de référence en ligne.

Pour MongoDB, nous suggérons fortement de mettre en signet et au moins de parcourir le manuel officiel de MongoDB qui est disponible dans quelques formats différents et constamment mis à jour sur http://www.mongodb.org/display/DOCS/Manual. Bien que le manuel décrive l'interface JavaScript via l' utilitaire de console mongo par opposition à l'interface Python, la plupart des extraits de code devraient être facilement compris par un programmeur Python et plus ou moins portables vers PyMongo, bien que parfois avec un peu de légèreté. de travail.
De plus, le manuel MongoDB approfondit certains sujets avancés et techniques de mise en œuvre et d'administration de bases de données que ce qui est possible dans ce livre.

Pour le langage Python et la bibliothèque standard, vous pouvez utiliser la fonction help() dans l'interpréteur ou l' outil pydoc sur la ligne de commande pour obtenir la documentation API pour n'importe quelle méthode ou module. Par exemple:

```
chaîne pydoc
```

La dernière documentation sur le langage Python et l'API est également disponible pour une navigation en ligne sur http://docs.python.org/.

Le pilote PyMongo de 10gen dispose d'une documentation API disponible en ligne pour accompagner chaque relocation. Vous pouvez le trouver sur http://api.mongodb.org/python/. De plus, une fois le package de pilotes PyMongo installé sur votre système, une version récapitulative de la documentation de l'API devrait être disponible dans l'interpréteur Python via la fonction help() . En raison d'un problème avec l' outil virtualenv mentionné dans la section suivante, « py-doc » ne fonctionne pas dans un environnement virtuel. Vous devez plutôt exécuter python -m pydoc pymongo.

Installation de MongoDB

À des fins de développement, il est recommandé d'exécuter un serveur MongoDB sur votre machine locale. Cela vous permettra d'itérer rapidement et d'essayer de nouvelles choses sans craindre de détruire une base de données de production. De plus, vous pourrez développer avec MongoDB même sans connexion Internet.

En fonction de votre système d'exploitation, vous pouvez disposer de plusieurs options pour installer MongoDB localement.

La plupart des systèmes modernes de type UNIX auront une version de MongoDB disponible dans leur système de gestion de packages. Cela inclut FreeBSD, Debian, Ubuntu, Fedora, Cen-tOS et ArchLinux. L'installation de l'un de ces packages est probablement l'approche la plus pratique, bien que la version de MongoDB fournie par votre fournisseur de package puisse être en retard par rapport à la dernière version de 10gen. Pour le développement local, tant que vous disposez de la dernière version majeure, tout va probablement bien. 10gen

fournit également ses propres packages MongoDB pour de nombreux systèmes qu'ils mettent à jour très rapidement à chaque version. Leur installation peut demander un peu plus de travail, mais assurez-vous que vous utilisez la version la plus récente et la plus performante. Après la configuration initiale, il est généralement simple de les maintenir à jour. Pour un déploiement de production, où vous souhaiterez probablement pouvoir mettre à jour vers la version stable la plus récente de MongoDB avec un minimum de tracas, cette option est probablement la plus logique.

En plus des versions du package système de MongoDB, 10gen fournit des archives binaires zip et tar. Ceux-ci sont indépendants de votre gestionnaire de packages système et sont fournis en versions 32 bits et 64 bits pour OS X, Windows, Linux et Solaris. 10gen fournit également des distributions binaires de ce type construites de manière statique pour Linux, ce qui peut être votre meilleure option si vous êtes bloqué sur un système Linux ancien et hérité dépourvu de la libc moderne.

et d'autres versions de bibliothèque. De plus, si vous utilisez OS X, Windows ou Solaris, ce sont probablement votre meilleur choix.

Enfin, vous pouvez toujours créer vos propres binaires à partir du code source. À moins que vous n'ayez besoin d'apporter vous-même des modifications aux composants internes de MongoDB, il est préférable d'éviter cette méthode en raison du temps et de la complexité impliqués.

Dans un souci de simplicité, nous fournirons les commandes nécessaires pour installer une version stable de MongoDB à l'aide du gestionnaire de packages système des systèmes d'exploitation de type UNIX les plus courants. C'est la méthode la plus simple, en supposant que vous soyez sur l'une de ces plateformes. Pour Mac OS X et Windows, nous fournissons des instructions pour installer les packages binaires de 10gen.

Ubuntu/Debian :

```
sudo apt-get update ; sudo apt-get install mongodb
```

Feutre:

```
sudo miam install mongo-stable-server
```

FreeBSD :

```
sudo pkg_add -r mongodb
```

Les fenêtres:

Allez sur http://www.mongodb.org et téléchargez le dernier fichier zip de la version de production pour Windows, en choisissant 32 bits ou 64 bits en fonction de votre système. Extrayez le contenu du fichier zip dans un emplacement tel que C:\mongodb et ajoutez le répertoire bin à votre PATH.

Mac OS X:

Allez sur http://www.mongodb.org et téléchargez le dernier fichier tar compressé de la version de production pour OS X, en choisissant 32 bits ou 64 bits en fonction de votre système. Extrayez le contenu vers un emplacement tel que /usr/local/ ou /opt et ajoutez le répertoire bin à votre $PATH.

Par exemple :

```
cd /tmp
wget http://fastdl.mongodb.org/osx/mongodb-osx-x86_64-1.8.3-rc1.tgz tar
xfz mongodb-osx-x86_64-1.8.3-rc1.tgz sudo
mkdir / usr/local/mongodb sudo
cp -r mongodb-osx-x86_64-1.8.3-rc1/bin /usr/local/mongodb/ export
PATH=$PATH:/usr/local/mongodb/bin
```

Installez MongoDB sur OS X avec les ports Mac Si

vous souhaitez essayer un système de gestion de packages système tiers sur Mac OS X, vous pouvez également installer MongoDB (et Python, en fait) via les ports Mac. Les ports Mac sont similaires aux ports FreeBSD, mais pour OS X.

Un mot d'avertissement cependant : Mac Ports se compile à partir des sources, et l'installation du logiciel peut donc prendre beaucoup plus de temps que la simple récupération des fichiers binaires. De plus, vous devrez installer les outils de développement Xcode d'Apple, ainsi que l'environnement de fenêtrage X11.

La première étape consiste à installer les ports Mac à partir de http://www.macports.org. Nous vous recommandons de télécharger et d'installer leur package DMG.

Une fois les ports Mac installés, vous pouvez installer MongoDB avec la commande :

```
mise à jour automatique du port sudo ; port sudo installer mongodb
```

Pour installer Python 2.7 à partir des ports Mac, utilisez la commande :

```
mise à jour automatique du port sudo ; port sudo installer python27
```

Exécuter MongoDB Sur

certaines plates-formes, telles qu'Ubuntu, le gestionnaire de paquets démarrera automatiquement le démon mongod pour vous et veillera à ce qu'il démarre également au démarrage. Sur d'autres, comme Mac OS X, vous devez écrire votre propre script pour le démarrer et l'intégrer manuellement à launchd afin qu'il démarre au démarrage du système.

Notez qu'avant de pouvoir démarrer MongoDB, ses répertoires de données et de journaux doivent exister.

Si vous souhaitez que MongoDB démarre automatiquement au démarrage sous Windows, 10gen dispose d'un document décrivant comment le configurer sur http://www.mongodb.org/display/DOCS/ Windows+Service

Pour que MongoDB démarre automatiquement au démarrage sous Mac OS X, vous aurez d'abord besoin d'un fichier plist. Enregistrez les éléments suivants (en modifiant les chemins de base de données et de journal de manière appropriée) dans /Library/LaunchDaemons/org.mongodb.mongod.plist :

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE
plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/ PropertyList -1.0.dtd"> <plist version="1.0">
<dict>


        <key>RunAtLoad</key>
        <true/>
        <key>Label</key>
        <string>org.mongo.mongod</string>
        <key>ProgramArguments</key>
        <array>
                <string>/usr/local/mongodb/bin/mongod</string> <string>--dbpath</
                string>
```

```
                <string>/usr/local/mongodb/data/</string> <string>--
                logpath</string> <string>/usr/
                local/mongodb/log/mongodb.log</string> </array> < /dict> </
plist>
```

Exécutez ensuite les commandes suivantes pour activer le script de démarrage avec launchd :

```
sudo launchctl load /Library/LaunchDaemons/org.mongodb.mongod.plist sudo
launchctl start org.mongodb.mongod
```

Un moyen rapide de tester si une instance MongoDB est déjà en cours d'exécution sur votre ordinateur local consiste à taper mongo sur la ligne de commande. Cela démarrera la console d'administration MongoDB, qui tente de se connecter à un serveur de base de données exécuté sur le port par défaut (27017).

Dans tous les cas, vous pouvez toujours démarrer MongoDB manuellement à partir de la ligne de commande. C'est une chose utile à connaître au cas où vous souhaiteriez tester des fonctionnalités telles que des jeux de répliques ou le partitionnement en exécutant plusieurs instances de Mongod sur votre machine locale.

En supposant que le binaire mongod se trouve dans votre $PATH, exécutez :

```
mongod --logpath <chemin/vers/mongo.logfile> --port <port sur lequel écouter> --dbpath <chemin/vers/
répertoire de données>
```

Configuration d'un environnement Python avec MongoDB

Afin de pouvoir vous connecter à MongoDB avec Python, vous devez installer le package de pilotes Py-Mongo. En Python, la meilleure pratique consiste à créer ce que l'on appelle un « environnement virtuel » dans lequel installer vos packages. Cela les isole proprement de tous les packages « système » que vous avez installés et offre l'avantage supplémentaire de ne pas nécessiter de privilèges root pour installer des packages Python supplémentaires. L'outil permettant de créer un « environnement virtuel » s'appelle virtualenv.

Il existe deux approches pour installer l'outil virtualenv sur votre système : manuellement et via votre outil de gestion des packages système. La plupart des systèmes modernes de type UNIX auront l'outil virtualenv dans leurs référentiels de packages. Par exemple, sur Mac OS X avec les ports Mac, vous pouvez exécuter sudo port install py27-virtualenv pour installer virtualenv pour Python 2.7. Sur Ubuntu, vous pouvez exécuter sudo apt-get install python-virtualenv. Reportez-vous à la documentation de votre système d'exploitation pour savoir comment l'installer sur votre plate-forme spécifique.

Si vous ne parvenez pas ou ne souhaitez tout simplement pas utiliser le gestionnaire de packages de votre système, vous pouvez toujours l'installer vous-même, à la main. Pour l'installer manuellement, vous devez disposer du package Python setuptools. Vous disposez peut-être déjà d'outils de configuration sur votre système. Vous pouvez tester cela en exécutant python -c import setuptools sur la ligne de commande. Si rien n'est imprimé et que vous revenez simplement à l'invite, vous n'avez rien à faire.
Si une ImportError est générée, vous devez installer setuptools.

Pour installer manuellement setuptools, téléchargez d'abord le fichier http://peak.telecommunity.com/dist/ez_setup.py Ensuite,

exécutez python ez_setup.py en tant que root.

Pour Windows, téléchargez et installez d'abord le dernier package Python 2.7.x à partir de http://www.python.org . Une fois que vous avez installé Python, téléchargez et installez le package d'installation de Windows setuptools à partir de http://pypi.python.org/pypi/setuptools/. Après avoir installé Python 2.7 et setuptools, l'outil easy_install sera disponible sur votre machine dans le répertoire des scripts Python (la valeur par défaut est C:\Python27\Scripts\).

Une fois que setuptools est installé sur votre système, exécutez easy_install virtualenv en tant que racine.

Maintenant que vous disposez de l'outil « virtualenv » sur votre machine, vous pouvez créer votre premier environnement Python virtuel. Vous pouvez le faire en exécutant la commande virtualenv --no-site-packages myenv. Vous n'avez pas besoin (et ne devriez en fait pas vouloir) d'exécuter cette commande avec les privilèges root. Cela créera un environnement virtuel dans le répertoire « myenv ». L'option --no-site-packages de l'utilitaire « virtualenv » lui demande de créer un environnement Python propre, isolé de tous les packages existants installés dans le système.

Vous êtes maintenant prêt à installer le pilote PyMongo.

Avec le répertoire « myenv » comme répertoire de travail (c'est-à-dire après « cd myenv »), exécutez simplement bin/easy_install pymongo. Cela installera la dernière version stable de PyMongo dans votre environnement Python virtuel. Pour vérifier que cela a fonctionné correctement, exécutez la commande bin/python -c import pymongo, en vous assurant que le répertoire « myenv » est toujours votre répertoire de travail, comme avec la commande précédente.

En supposant que Python n'a pas généré d'ImportError, vous disposez maintenant d'un environnement virtuel Python avec le pilote PyMongo correctement installé et êtes prêt à vous connecter à MongoDB et à commencer à émettre des requêtes !

CHAPITRE 2

Lire et écrire vers MongoDB avec Python

MongoDB est une base de données orientée document. Ceci est différent d'une base de données relationnelle de deux manières significatives. Premièrement, toutes les entrées ne doivent pas nécessairement respecter le même schéma. Deuxièmement, vous pouvez intégrer des entrées les unes dans les autres. Malgré ces différences majeures, il existe des analogues aux concepts SQL dans MongoDB. Un groupe logique d'entrées dans un SQL la base de données est appelée une table. Dans MongoDB, le terme analogue est une collection. Un célibataire ou Individual une entrée dans une base de données SQL est appelée une ligne. Dans MongoDB, l'analogue est un document.

Tableau 2-1. Comparaison des concepts et termes SQL/SGBDR et MongoDB

Concept	SQL	MongoDB
Un utilisateur	Une rangée	Un document
Tous les utilisateurs	Tableau des utilisateurs	Collecte des utilisateurs
Un nom d'utilisateur par utilisateur (1 pour 1)	Colonne Nom d'utilisateur	Propriété du nom d'utilisateur
Plusieurs e-mails par utilisateur (1 à plusieurs)	SQL JOIN avec la table des e-mails	Intégrer le document électronique pertinent dans l'utilisateur Document
De nombreux éléments appartenant à de nombreux utilisateurs (plusieurs à plusieurs)	SQL JOIN avec la table des éléments	Rejoindre par programme des éléments Collection

Ainsi, dans MongoDB, vous opérez principalement sur des documents et des collections de documents. Si vous êtes familier avec JSON, un document MongoDB est essentiellement un JSON document avec quelques fonctionnalités supplémentaires. Du point de vue de Python, il s'agit d'un dictionnaire Python.

Prenons l'exemple suivant d'un document utilisateur avec un nom d'utilisateur, un prénom, un nom, une date de naissance, une adresse e-mail et un score :

```
à partir de dateheure importer dateheure
utilisateur_doc = {
    "nom d'utilisateur" : "janedoe",
    "prénom" : "Jeanne",
```

```
        "nom" : "Doe",
        "dateofbirth" : datetime(1974, 4, 12), "email" :
        "janedoe74@example.com", "score" : 0

}
```

Comme vous pouvez le voir, il s'agit d'un objet Python natif. Contrairement à SQL, il n'y a pas de syntaxe particulière à gérer. Le pilote PyMongo prend en charge de manière transparente les objets datetime Python. Ceci est très pratique lorsque vous travaillez avec des instances datetime : le pilote rassemblera les valeurs de manière transparente pour vous en lecture et en écriture. Vous ne devriez jamais avoir à écrire vous-même le code de conversion datetime.

Au lieu de regrouper des éléments dans des tables, comme en SQL, MongoDB les regroupe dans des collections. Comme les tables SQL, les collections MongoDB peuvent avoir des index sur des propriétés de document particulières pour des recherches plus rapides et vous pouvez y lire et écrire à l'aide de prédicats de requête complexes. Contrairement aux tables SQL, les documents d'une collection MongoDB ne doivent pas tous être conformes au même schéma.

En revenant à notre exemple d'utilisateur ci-dessus, ces documents seraient logiquement regroupés dans une collection « utilisateurs ».

Connexion à MongoDB avec Python

Le pilote PyMongo facilite la connexion à une base de données MongoDB.
De plus, le pilote prend en charge certaines fonctionnalités intéressantes dès la sortie de la boîte, telles que le regroupement de connexions et la reconnexion automatique en cas d'échec (lorsque vous travaillez avec une configuration répliquée). Si vous êtes familier avec les systèmes SGBDR/SQL plus traditionnels, par exemple MySQL, vous êtes probablement habitué à déployer des logiciels supplémentaires, voire même à écrire les vôtres, pour gérer le regroupement de connexions et la reconnexion automatique. 10gen nous a très soigneusement soulagé de la nécessité de nous soucier de ces détails lorsque nous travaillons avec MongoDB et le pilote PyMongo. Cela simplifie grandement l'exécution d'un système de production basé sur MongoDB.

Vous instanciez un objet Connection avec les paramètres nécessaires. Par défaut, l'objet Connection se connectera à un serveur MongoDB sur localhost sur le port 27017. Pour être explicite, nous transmettrons ces paramètres dans notre exemple :

```
    """                                                    """
        Un exemple de connexion au système d'importation
    MongoDB

    depuis pymongo import Connection depuis
    pymongo.errors import ConnectionFailure

    def main() :                                """
            Connectez-vous à

        MongoDB essayez : c = Connection(host="localhost", port=27017)
            print "Connecté avec succès" sauf
        ConnectionFailure, e :
            sys.stderr.write("Impossible de se connecter à MongoDB : %s" % e)
```

```
        sys.exit(1)

si __name__ == "__main__":
    main()
```

Comme vous pouvez le voir, une exception ConnectionFailure peut être levée par l'instanciation de Connection. C'est généralement une bonne idée de gérer cette exception et de fournir quelque chose d'informatif à vos utilisateurs.

Obtenir un handle de base de données

Les objets de connexion eux-mêmes ne sont pas très fréquemment utilisés lorsque vous travaillez avec MongoDB en Python. En général, vous en créez un une fois, puis vous l'oubliez. En effet, la plupart des interactions réelles se produisent avec les objets Database et Collection.
Les objets de connexion ne sont qu'un moyen de maîtriser votre premier objet Databse. En fait, même si vous perdez la référence à l'objet Connection, vous pouvez toujours la récupérer car les objets Database ont une référence à l'objet Connection.

Obtenir un objet Database est facile une fois que vous disposez d'une instance de Connection. Vous avez simplement besoin de connaître le nom de la base de données, ainsi que le nom d'utilisateur et le mot de passe pour y accéder si vous utilisez une autorisation sur celle-ci.

```
    """
        Un exemple de comment obtenir un handle Python vers un système d'importation de base de
    données MongoDB

    depuis pymongo import Connection
    depuis pymongo.errors import ConnectionFailure

    def main() :
        """
            Connectez-vous à

        MongoDB essayez : c = Connection(host="localhost",
        port=27017) sauf ConnectionFailure, e :
            sys.stderr.write("Impossible de se connecter à MongoDB : %s" %
            e) sys.exit(1)
        # Récupère un handle de base de données vers une base de données nommée
        "mydb" dbh = c["mydb"]

        # Démontrez la propriété db.connection pour récupérer une référence à l'objet # Connection s'il sort
        de la portée. Dans la plupart des cas, conserver une # référence à l'objet Database pendant toute la
        durée de vie de votre programme devrait # être suffisant.


        assert dbh.connection == c print
        "Configuration réussie d'un handle de base de données"

si __name__ == "__main__":
    main()
```

Insérer un document dans une collection

Une fois que vous maîtrisez votre base de données, vous pouvez commencer à insérer des données. Imaginons que nous ayons une collection appelée « utilisateurs », contenant tous les utilisateurs de notre jeu. Chaque utilisateur dispose d'un nom d'utilisateur, d'un prénom, d'un nom, d'une date de naissance, d'une adresse email et d'un score. Nous souhaitons ajouter un nouvel utilisateur :

```
"""                                                                    """
    Un exemple de comment insérer un système d'importation
de documents

depuis datetime import datetime
depuis pymongo import Connection
depuis pymongo.errors import ConnectionFailure

def main() :

    essayez : c = Connection(host="localhost", port=27017) sauf
    ConnectionFailure, e :
        sys.stderr.write("Impossible de se connecter à MongoDB : %s" % e) sys.exit(1)
        dbh = c["mydb"]
    assert dbh.connection
    == c user_doc = { "username" :
    "janedoe" ,
        "prénom" : "Jane", "nom" : "Doe",
        "dateofbirth" : datetime(1974,
        4, 12), "email" :
        "janedoe74@example.com", "score" : 0


    }

    dbh.users.insert(user_doc, safe=True) print
    "Document inséré avec succès : %s" % user_doc

si __name__ == "__main__":
    main()
```

Notez que nous n'avons pas besoin de dire à MongoDB de créer notre collection « utilisateurs » avant de l'insérer. Les collections sont créées paresseusement dans MongoDB, chaque fois que vous y accédez. Cela a l'avantage d'être très léger, mais peut parfois poser des problèmes dus à des fautes de frappe. Ceux-ci peuvent être difficiles à retrouver à moins que vous ne disposiez d'une bonne couverture de tests. Par exemple, imaginez que vous ayez tapé accidentellement :

```
# dbh.usrs est une faute de frappe, nous voulons dire dbh.users ! Contrairement à un
SGBDR, MongoDB ne vous protégera pas de ce
type d'erreur. dbh.usrs.insert(user_doc)
```

Le code s'exécuterait correctement et aucune erreur ne serait générée. Vous pourriez vous demander pourquoi votre document utilisateur n'est pas là. Nous vous recommandons d'être extrêmement vigilant et de vérifier votre orthographe lorsque vous abordez des collections. Une bonne couverture des tests peut également aider à détecter des bugs de ce type.

Une autre fonctionnalité des inserts MongoDB à prendre en compte est la génération automatique de clé primaire. Dans MongoDB, la propriété _id sur un document est traitée spécialement. Il est considéré comme la clé primaire de ce document et devrait être unique à moins que la collection a été explicitement créé sans index sur _id. Par défaut, si aucune propriété _id n'est présent dans un document que vous insérez, MongoDB en générera un lui-même. Quand MongoDB génère lui-même une propriété _id , il utilise le type ObjectId. Un ObjectId MongoDB est un Valeur de 96 bits qui devrait avoir une très forte probabilité d'être unique lorsque créé. Son objectif peut être considéré comme similaire à un objet UUID tel que défini par RFC. 4122. Les ObjectIds MongoDB ont la belle propriété d'être presque certainement uniques au fil de la génération, aucune coordination centrale n'est donc requise.

Cela contraste fortement avec le langage courant des SGBDR consistant à utiliser des clés primaires à incrémentation automatique. Pour garantir qu'une clé d'auto-incrémentation n'est pas déjà utilisée, il faut généralement consulter un système centralisé. Lorsque l'intention est de fournir une base de données évolutive horizontalement, décentralisée et tolérante aux pannes (comme c'est le cas avec Mon-goDB), les clés à incrémentation automatique représentent un vilain goulot d'étranglement.

En utilisant ObjectId comme _id, vous laissez la porte ouverte à la mise à l'échelle horizontale via Capacités de partitionnement de MongoDB. Bien que vous puissiez en fait fournir votre propre valeur pour le _id si vous le souhaitez (à condition qu'elle soit globalement unique), il est préférable d'éviter cela à moins que il y a de bonnes raisons de faire autrement. Exemples de cas où vous pourriez être contraint pour fournir votre propre valeur de propriété _id , incluez la migration à partir des systèmes SGBDR qui a utilisé l'idiome de clé primaire à incrémentation automatique mentionné précédemment.

Notez qu'un ObjectId peut être tout aussi facilement généré côté client, avec PyMongo, comme par le serveur. Pour générer un ObjectId avec PyMongo, vous instanciez simplement pymongo.objectid.ObjectId.

Écrivez dans une collection en toute sécurité et de manière synchrone

Par défaut, le pilote PyMongo effectue des écritures asynchrones. Les opérations d'écriture incluent l'insertion, la mise à jour, la suppression et la recherche et la modification.

Les écritures asynchrones ne sont pas sécurisées dans le sens où elles ne sont pas vérifiées pour détecter les erreurs et Ainsi, l'exécution de votre programme pourrait continuer sans aucune garantie que l'écriture se soit terminée avec succès. Bien que les écritures asynchrones améliorent les performances en ne bloquant l'exécution, ils peuvent facilement conduire à des conditions de concurrence désagréables et à d'autres bogues d'intégrité des données. Pour cette raison, nous vous recommandons d'utiliser presque toujours des écritures bloquantes, synchronisées et sécurisées. Il semble rare en pratique d'avoir des écritures véritablement « feu-et-oubli » où il n'y a absolument aucune conséquence en cas d'échec. Cela étant dit, un point commun Un exemple où les écritures asynchrones peuvent avoir du sens est lorsque vous écrivez des journaux ou des données d'analyse non critiques dans MongoDB à partir de votre application.

Sauf si vous êtes certain de ne pas avoir besoin d'écritures synchrones, nous vous recommandons de transmettre l'argument de mot-clé « safe=True » aux opérations d'insertion, de mise à jour, de suppression et de findAndModify :

```
# safe=True garantit que votre #
écriture réussira ou qu'une exception sera levée
dbh.users.insert(user_doc, safe=True)
```

Garantir les écritures sur plusieurs nœuds de base de données

Le terme nœud fait référence à une instance unique du processus démon MongoDB. Il existe généralement un seul nœud MongoDB par machine, mais pour les cas de test ou de développement, vous pouvez exécuter plusieurs nœuds sur une seule machine.

Replica Set est le terme MongoDB désignant la configuration améliorée de réplication maître-esclave de la base de données. Ceci est similaire à la réplication maître-esclave traditionnelle que vous trouvez dans les SGBDR tels que MySQL et PostgreSQL dans la mesure où un seul nœud gère les écritures à un moment donné. Dans MongoDB, la sélection du maître est déterminée par un protocole d'élection et lors du basculement, un esclave est automatiquement promu maître sans nécessiter l'intervention de l'opérateur. De plus, le pilote PyMongo est compatible avec Replica Set et effectue une reconnexion automatique en cas d'échec au nouveau maître. Les ensembles de réplicas MongoDB représentent donc une configuration de réplication maître-esclave avec une excellente gestion des pannes dès le départ. Pour tous ceux qui ont dû effectuer une récupération manuelle après une panne de maître MySQL dans un environnement de production, cette fonctionnalité est un soulagement bienvenu.

Par défaut, MongoDB renverra la réussite de votre opération d'écriture une fois qu'elle aura été écrite sur un seul nœud dans un jeu de réplicas.

Cependant, pour plus de sécurité en cas d'échec, vous souhaiterez peut-être que votre écriture soit validée sur deux répliques ou plus avant de renvoyer un succès. Cela peut permettre de garantir qu'en cas de panne catastrophique, au moins un des nœuds du jeu de réplicas disposera de votre écriture.

PyMongo permet de spécifier facilement le nombre de nœuds sur lesquels vous souhaitez que votre écriture soit répliquée avant de réussir. Vous définissez simplement un paramètre nommé « w » sur le nombre de serveurs dans chaque appel de méthode d'écriture.

Par exemple:

```
# w=2 signifie que l'écriture ne réussira pas tant qu'elle # n'aura pas été
écrite sur au moins 2 serveurs dans un jeu de répliques.
dbh.users.insert(user_doc, w=2)
```

Notez que transmettre n'importe quelle valeur de « w » à une méthode d'écriture dans PyMongo implique également de définir « safe=True ».

Introduction au langage de requête MongoDB

Les requêtes MongoDB sont représentées sous la forme d'une structure de type JSON, tout comme les documents. Pour créer une requête, vous spécifiez un document avec les propriétés auxquelles vous souhaitez que les résultats correspondent. MongoDB traite chaque propriété comme ayant un ET booléen implicite. Il prend nativement en charge les requêtes booléennes OR, mais vous devez utiliser un opérateur spécial ($or) pour y parvenir. En plus des correspondances exactes, MongoDB propose des opérateurs pour supérieur à, inférieur à, etc.

Exemple de document de requête pour faire correspondre tous les documents de la collection des utilisateurs portant le prénom « Jane » :

```
q =
    { "prénom" : "jane"
}
```

Si nous voulions récupérer tous les documents avec le prénom « jane » ET le nom « doe », nous écririons :

```
q =
    { "prénom" : "jane", "nom" :
      "biche"
}
```

Si l'on voulait récupérer tous les documents ayant une valeur de score supérieure à 0, on écrirait :

```
q =
    { "score" : { "$gt" : 0 }
}
```

Notez l'utilisation de l'opérateur spécial « $gt ». Le langage de requête MongoDB fournit un certain nombre de ces opérateurs, vous permettant de créer des requêtes assez complexes.

Voir la section sur les opérateurs de requête MongoDB pour plus de détails.

Lire, compter et trier les documents dans une collection

Dans de nombreuses situations, vous souhaitez uniquement récupérer un seul document d'une collection. Cela est particulièrement vrai lorsque les documents de votre collection sont uniques sur certaines propriétés. Un bon exemple de ceci est une collection d'utilisateurs, où chaque nom d'utilisateur est garanti unique.

```
# En supposant que nous ayons déjà un handle de base de données nommé dbh
dans la portée #, trouvez un seul document avec le nom d'utilisateur
"janedoe". user_doc = dbh.users.find_one({"username" : "janedoe"})
sinon user_doc :
    imprimer "aucun document trouvé pour le nom d'utilisateur janedoe"
```

Notez que find_one() renverra None si aucun document n'est trouvé.

Imaginez maintenant que vous souhaitiez rechercher tous les documents de la collection des utilisateurs dont la propriété prénom est définie sur « jane » et imprimer leurs adresses e-mail. MongoDB nous renverra un objet Cursor pour diffuser les résultats. PyMongo gère le streaming des résultats

au fur et à mesure que vous itérez, donc si vous avez un grand nombre de résultats, ils ne sont pas tous stockés en mémoire immediatement.

```
# En supposant que nous ayons déjà un handle de base de données dans la portée nommé
dbh # trouver tous les documents portant le prénom "jane".
# Ensuite, parcourez-les et imprimez l'adresse e-mail. users =
dbh.users.find({"firstname":"jane"}) pour l'utilisateur dans les
utilisateurs :
    print user.get("email")
```

Notez dans l'exemple ci-dessus que nous utilisons la méthode get de la classe Python dict . Si nous étions certains que chaque document de résultat contenait la propriété « email », nous aurions pu utiliser l'accès au dictionnaire à la place.

```
pour l'utilisateur dans les utilisateurs :
    imprimer l'utilisateur["email"]
```

Si vous souhaitez uniquement récupérer un sous-ensemble des propriétés de chaque document d'une collection lors d'une lecture, vous pouvez les transmettre sous forme de dictionnaire via un paramètre supplémentaire. Par exemple, supposons que vous souhaitiez récupérer uniquement l'adresse email de chaque utilisateur portant le prénom « jane » : # Récupérez uniquement le champ « email » de

```
chaque document correspondant. users = dbh.users.find({"firstname":"jane"}, {"email":1})
pour l'utilisateur dans les utilisateurs : print user.get("email")
```

Si vous récupérez un jeu de résultats volumineux, demander uniquement les propriétés dont vous avez besoin peut réduire la surcharge du réseau et du décodage, augmentant ainsi potentiellement les performances.

Parfois, vous n'êtes pas tellement intéressé par les résultats de la requête eux-mêmes, mais cherchez à trouver la taille de l'ensemble de résultats pour une requête donnée. Un exemple courant est une situation d'analyse dans laquelle vous souhaitez connaître le nombre de documents présents dans les collections de vos utilisateurs.
MonogDB prend en charge un comptage efficace des jeux de résultats côté serveur avec la méthode count() sur les objets Cursor : # Découvrez

```
efficacement combien de documents se trouvent dans la collection des utilisateurs
userscount = dbh.users.find().count() print "Il y a %
d documents dans la collection des utilisateurs" % nombre d'utilisateurs
```

MongoDB peut également effectuer le tri des résultats pour vous côté serveur. Surtout si vous triez les résultats sur une propriété qui a un index, il peut les trier beaucoup plus efficacement que votre programme client. Les objets PyMongo Cursor ont une méthode sort() qui prend un tuple Python 2 comprenant la propriété sur laquelle trier et la direction. La méthode Py-Mongo sort() est analogue à l'instruction SQL ORDER BY. La direction peut être pymongo.ASCENDING ou pymongo.DESCENDING. Par exemple:

```
# Renvoie tous les utilisateurs avec le prénom "jane" triés # par ordre
décroissant par date de naissance (c'est-à-dire le plus jeune en premier) users =

dbh.users.find( {"firstname":"jane"}).sort(("dateofbirth", pymongo.DESCENDING )) pour l'utilisateur
dans les utilisateurs : print
    user.get("email")
```

En plus de la méthode sort() sur l'objet PyMongo Cursor, vous pouvez également transmettre des instructions de tri aux méthodes find() et find_one() sur l'objet PyMongo Collection. En utilisant cette fonctionnalité, l'exemple ci-dessus peut être réécrit comme suit :

```
# Renvoie tous les utilisateurs portant le prénom "jane" triés # par
ordre décroissant par date de naissance (c'est-à-dire le plus jeune en premier)
users = dbh.users.find({"firstname":"jane"},
    sort=[("dateofbirth", pymongo.DESCENDING)]) pour
l'utilisateur dans les
    utilisateurs : print user.get("email")
```

Une autre situation que vous pouvez rencontrer, en particulier lorsque vous disposez de grands ensembles de résultats, est que vous souhaitez récupérer uniquement un nombre limité de résultats. Ceci est fréquemment combiné avec un tri des résultats côté serveur. Par exemple, imaginez que vous générez un tableau des meilleurs scores qui affiche uniquement les dix meilleurs scores. Les objets PyMongo Cursor ont une méthode limit() qui permet cela. La méthode limit() est analogue à l'état SQL LIMIT.
ment.

```
# Renvoie au maximum 10 utilisateurs triés par score dans l'ordre décroissant # Ceci
peut être utilisé comme "tableau des meilleurs scores des 10 meilleurs
utilisateurs" users = dbh.users.find().sort(("score", pymongo.DESCENDING)).limit (10) pour
l'utilisateur dans les
    utilisateurs : print user.get("username"), user.get("score", 0)
```

Si vous savez à l'avance que vous n'avez besoin que d'un nombre limité de résultats d'une requête, l'utilisation de limit() peut apporter un avantage en termes de performances. En effet, cela peut réduire considérablement la taille des données de résultats qui doivent être envoyées par MongoDB. Notez qu'une limite de 0 équivaut à aucune limite.

De plus, MongoDB peut prendre en charge le passage à un décalage spécifique dans un jeu de résultats via la méthode Cursor.skip() fournie par PyMongo. Lorsqu'il est utilisé avec limit() , cela permet la pagination des résultats qui est fréquemment utilisée par les clients lorsqu'ils permettent aux utilisateurs finaux de parcourir de très grands ensembles de résultats. skip() est analogue à l'instruction SQL OFFSET. Par exemple, imaginez une application Web qui affiche 20 utilisateurs par page, triés par ordre alphabétique et qui doit récupérer les nom de famille , données pour créer la deuxième page de résultats pour un utilisateur. Le
La requête utilisée par l'application Web peut ressembler à ceci :

```
# Renvoie au maximum 20 utilisateurs triés par nom, # en
ignorant les 20 premiers résultats dans l'ensemble users =
dbh.users.find().sort( ("nom",
    pymongo.ASCENDING)).limit(20).skip(20 )
```

Enfin, lorsque vous parcourez de très grands ensembles de résultats, où les documents sous-jacents peuvent être modifiés par d'autres programmes en même temps, vous souhaiterez peut-être utiliser le mode Snap-shot de MongoDB. Imaginez un site très fréquenté avec des centaines de milliers d'utilisateurs. Vous développez un programme d'analyse pour compter les utilisateurs et créer diverses statistiques sur les modèles d'utilisation, etc. Cependant, ce programme d'analyse est destiné à être exécuté sur la base de données de production en direct. Comme il s'agit d'un site très fréquenté, les utilisateurs réels effectuent fréquemment des actions sur le site, ce qui peut entraîner des modifications de leurs documents utilisateur correspondants, pendant que votre programme d'analyse est en cours d'exécution. En raison de bizarreries dans le programme actuel de MongoDB-

mécanisme de tri, dans ce genre de situation, votre programme pourrait facilement voir les doublons dans le jeu de résultats de votre requête. Les données en double pourraient nuire à la précision de votre programme d'analyse, il est donc préférable de les éviter. C'est là qu'intervient le mode instantané.

Le mode Snapshot de MongoDB garantit que les documents modifiés pendant la durée de vie d'une requête ne sont renvoyés qu'une seule fois dans un jeu de résultats. En d'autres termes, les doublons sont éliminés et vous ne devriez pas avoir à vous en soucier.

Cependant, le mode Snapshot présente certaines limites. Le mode Snapshot ne peut pas être utilisé avec le tri, ni avec un index sur une propriété autre que _id.

Pour utiliser le mode Snapshot avec PyMongo, passez simplement « snapshot=True » comme paramètre à la méthode find() :

```
# Parcourez toute la collection d'utilisateurs, en utilisant le mode Snapshot
# pour éliminer les résultats potentiels en double.
pour l'utilisateur dans dbh.users.find(snapshot=True) :
    print user.get("nom d'utilisateur"), user.get("score", 0)
```

Mise à jour de documents dans une collection

Les requêtes de mise à jour dans MongoDB se composent de deux parties : une spécification de document qui informe la base de données de l'ensemble de documents à mettre à jour et le document de mise à jour lui-même.

La première partie, la spécification du document, est la même que le document de requête que vous utilisez avec find() ou find_one().

La deuxième partie, le document de mise à jour, peut être utilisée de deux manières. Le plus simple est de fournir le document complet qui remplacera le document correspondant dans la collection.
Par exemple, supposons que vous ayez le document suivant dans votre collection d'utilisateurs :

```
user_doc =
    { "username" : "janedoe",
    "firstname" : "Jane", "nom" :
    "Doe", "dateofbirth" :
    datetime(1974, 4, 12), "email" : "janedoe74@example.com
    ", "note" : 0

}
```

Supposons maintenant que vous souhaitiez mettre à jour le document avec le nom d'utilisateur « janedoe » pour changer l'adresse e-mail en « janedoe74@example2.com ». Nous construisons un tout nouveau document identique à l'original, à l'exception de la nouvelle adresse e-mail.

```
# première requête pour obtenir une copie de la copie
d'importation
du document actuel old_user_doc =
dbh.users.find_one({"username":"janedoe"})
new_user_doc = copy.deepcopy(old_user_doc) #
modifier la copie pour changer l'adresse e-mail
new_user_doc[ "email"] =
"janedoe74@example2.com" # exécute la requête de mise à jour # remplace
le document correspondant par le contenu de new_user_doc dbh.users.update({"username":"janedoe"}, new_user_doc, safe
```

La création de l'intégralité du document de remplacement peut s'avérer fastidieuse et, pire encore, peut introduire des conditions de concurrence. Imaginez que vous souhaitiez incrémenter la propriété score de l'utilisateur « jane-doe ». Pour y parvenir avec l'approche de remplacement, vous devrez d'abord récupérer le document, le modifier avec le score incrémenté, puis le réécrire dans la base de données. Avec cette approche, vous pourriez facilement perdre d'autres modifications de partition si quelque chose d'autre devait mettre à jour la partition entre votre lecture et votre écriture.

Afin de résoudre ce problème, le document de mise à jour prend en charge un ensemble supplémentaire d'opérateurs MongoDB appelés « modificateurs de mise à jour ». Ces modificateurs de mise à jour incluent des opérateurs tels que l'incrémentation/décrémentation atomique, le push/pop de liste atomique, etc. Il est très utile de savoir quels modificateurs de mise à jour sont disponibles et ce qu'ils peuvent faire lors de la conception de votre application. Beaucoup d'entre eux seront décrits dans leurs propres recettes tout au long de ce livre.

Pour illustrer l'utilisation des « modificateurs de mise à jour », revenons à notre exemple original consistant à modifier uniquement l'adresse e-mail du document avec le nom d'utilisateur « janedoe ». Nous pouvons utiliser le modificateur $set update dans notre document de mise à jour pour éviter d'avoir à interroger avant la mise à jour. $set modifie la valeur d'une propriété individuelle ou d'un groupe de propriétés selon ce que vous spécifiez.

```
# exécutez la requête de mise à jour, en utilisant le modificateur $set update. #
nous n'avons pas besoin de connaître le contenu actuel du document # avec cette approche,
et évitons ainsi une requête initiale et # une condition de concurrence potentielle.
dbh.users.update({"username":"janedoe"},
    {"$set":{"email":"janedoe74@example2.com"}}, safe=True)
```

Vous pouvez également définir plusieurs propriétés à la fois à l'aide du modificateur $set update :

```
# mettre à jour l'adresse email et le score en même temps # en
utilisant $set en une seule écriture.
dbh.users.update({"username":"janedoe"},
    {"$set":{"email":"janedoe74@example2.com", "score":1}}, safe=True)
```

Au moment de la rédaction, le pilote PyMongo, même si vous spécifiez une spécification de document à la méthode de mise à jour qui correspond à plusieurs documents d'une collection, applique uniquement la mise à jour au premier document correspondant.

En d'autres termes, même si vous pensez que les spécifications de votre document de mise à jour correspondent à tous les documents de la collection, votre mise à jour n'écrira que sur un seul de ces documents.

Par exemple, imaginons que nous souhaitions définir un indicateur sur chaque document de notre collection d'utilisateurs qui a un score de 0 :

```
# même si chaque document de votre collection a un score de 0, # seul le premier
document correspondant aura sa propriété "flaked" définie sur True. dbh.users.update({"score":0},{"$set":
{"flaged":True}}, safe=True)
```

Afin que votre requête de mise à jour écrive plusieurs documents, vous devez transmettre le paramètre « multi=True » à la méthode de mise à jour.

```
# une fois que nous avons fourni le paramètre "multi=True", tous les documents correspondants
# seront mis à jour
dbh.users.update({"score":0},{"$set":{"flagg":True}}, multi= Vrai, sûr = Vrai)
```

Bien que la valeur par défaut du paramètre multi de la méthode de mise à jour soit actuellement False (ce qui signifie que seul le premier document correspondant sera mis à jour), cela peut changer. La documentation PyMongo recommande actuellement de définir explicitement multi=False si vous comptez sur cette valeur par défaut, pour éviter toute casse à l'avenir. Notez que cela ne devrait vous affecter que si vous travaillez avec une collection dans laquelle vos documents ne sont pas uniques sur la propriété sur laquelle vous interrogez dans votre spécification de document.

Suppression de documents d'une collection

Si vous souhaitez supprimer définitivement des documents d'une collection, c'est assez simple. L'objet PyMongo Collection a une méthode Remove() . Comme pour les lectures et les mises à jour, vous spécifiez les documents que vous souhaitez supprimer au moyen d'une spécification de document. Par exemple, pour supprimer tous les documents de la collection des utilisateurs avec un score de 1, vous utiliserez le code suivant :

```
# Supprimez tous les documents de la collection utilisateur avec le score 1
dbh.users.remove({"score":1}, safe=True)
```

Notez que la méthode Remove() prend un paramètre sécurisé . Comme mentionné dans la section précédente « Écrire dans une collection de manière sûre et synchrone », il est recommandé de définir le paramètre safe sur True sur les méthodes d'écriture pour garantir que l'opération est terminée. Il convient également de noter que Remove() ne générera aucune exception ou erreur si aucun document ne correspond.

Enfin, si vous souhaitez supprimer tous les documents d'une collection, vous pouvez passer Aucun comme paramètre à supprimer() :

```
# Supprimer tous les documents de la collection utilisateur
dbh.users.remove (Aucun, safe = True)
```

Effacer une collection avec remove() diffère de la suppression de la collection via drop_col lection() dans la mesure où les index resteront intacts.

Opérateurs de requête MongoDB

Comme mentionné précédemment, MongoDB dispose d'un ensemble assez riche d'opérateurs de requête et de prédicats. Dans le tableau 2-2 , nous fournissons un tableau avec la signification de chacun, ainsi qu'un exemple d'utilisation et l'équivalent SQL, le cas échéant.

Tableau 2-2. Opérateurs de requête MongoDB

Opérateur	Signification	Exemple	Équivalent SQL
$gt	Plus grand que	"score":{"$gt":0}	>
$lt	Moins que	"score":{"$lt":0}	<
$gte	Meilleur que ou égal	"score":{"$gte":0}	>=
$lte	Inférieur ou égal	"score":{"$lte":0}	
$tout	Le tableau doit tout contenir	"compétences":{"$all":["mongodb","python"]}	N / A
$existe	La propriété doit exister	« e-mail » : {"$existe » : Vrai}	N / A
$mod	Module X est égal à Y	"secondes":{"$mod":[60,0]}	MOD()
$ne	Pas égal	"secondes":{"$ne":60}	!=
$ en	Dans	"compétences":{"$in":["c","c++"]}	DANS
$nin	Pas dedans	"compétences":{"$nin":["php","ruby","perl"]} "$nor":	PAS DEDANS
$ni	Ni	[{"langue":"english"},{"pays":"usa" }]	N / A
$ou	Ou	"$or":[{"langue":"english"},{"pays":"usa"}]	OU
$taille	Le tableau doit être de taille	« compétences » :{"$size":3}	N / A

Si vous ne comprenez pas complètement la signification ou le but de certains de ces opérateurs immédiatement ne vous inquiétez pas. Nous discuterons de l'utilisation pratique de certains des plus opérateurs avancés en détail au chapitre 3.

Modificateurs de mise à jour MongoDB

Comme mentionné dans la section « Mise à jour des documents dans une collection », MongoDB vient avec un ensemble d'opérateurs pour effectuer des modifications atomiques sur les documents.

Tableau 2-3. Modificateurs de mise à jour MongoDB

Modificateur	Signification	Exemple
$inc	Incrément atomique	"$inc":{"score":1}
$ensemble	Définir la valeur de la propriété	"$set":{"nom d'utilisateur":"niall"}
$non défini	Propriété Désactiver (supprimer)	"$unset":{"nom d'utilisateur":1}
$ pousser	Ajout d'un tableau atomique (atome)	"$push":{"e-mails":"foo@example.com"}
$pushTout	Ajout d'un tableau atomique (liste)	"$pushall":{"e-mails":["foo@example.com","foo2@ex-ample.com"]}
$ajouterToSet	Ajout atomique si non présent	"$addToSet":{"e-mails":"foo@example.com"}
$pop	Supprimer la queue du réseau atomique	"$pop":{"e-mails":1}
$ tirer	Élément de tableau conditionnel atomique Suppression	"$pull":{"e-mails":"foo@example.com"}
$pullTout	Retrait de plusieurs éléments de la matrice atomique	"$pullAll":{"emails":["foo@example.com", "foo2@ex-ample.com"]}
$renommer	Renommer la propriété atomique	"$rename":{"emails":"old_emails"}

Comme pour les opérateurs de requête MongoDB répertoriés plus haut dans ce chapitre, cette table est principalement pour votre référence. Ces opérateurs seront présentés plus en détail au chapitre 3.

CHAPITRE 3

MongoDB commun et Modèles Python

Après un certain temps de travail avec MongoDB et Python pour résoudre différents problèmes, divers modèles et meilleures pratiques commencent à émerger. Comme pour tout langage de programmation et système de base de données, il existe des approches établies pour modéliser les données ainsi que des méthodes connues pour répondre aux requêtes aussi rapidement et efficacement que possible.

Bien qu'il existe une myriade de sources de telles connaissances pour les systèmes RDBM traditionnels comme MySQL, il existe beaucoup moins de ressources disponibles pour MongoDB. Ce chapitre est une tentative de résoudre ce problème.

Un modèle unique orienté document : l'intégration

Bien que la capacité des documents MongoDB à contenir des sous-documents ait été mentionnée précédemment dans ce livre, elle n'a pas été explorée en détail. En fait, l'intégration est une technique de modélisation extrêmement importante lorsque l'on travaille avec MongoDB et peut avoir des implications importantes en termes de performances et d'évolutivité. En particulier, l'intégration peut être utilisée pour résoudre de nombreux problèmes de modélisation de données généralement résolus par une jointure dans les SGBDR traditionnels. De plus, l'intégration est peut-être plus intuitive et plus facile à comprendre qu'une jointure.

Qu'entend-on exactement par intégration ? En termes Python, lorsque la valeur d'une clé dans un dictionnaire est encore un autre dictionnaire, nous disons que vous intégrez cette dernière dans le premier. Par exemple:

```
mon_document
    = { "name": "foo
    document", "data": {"name": "bar document"}
}
```

Ici, « data » est un sous-document intégré dans le document de niveau supérieur « my_document ».

L'intégration de sous-documents peut être une technique utile et naturelle pour réduire l'encombrement ou les collisions d'espaces de noms. Par exemple, considérons le cas où un document « utilisateur » doit faire référence aux noms d'utilisateur, aux mots de passe et aux détails associés des comptes Facebook, Twitter et IRC, en plus de stocker une propriété « nom d'utilisateur » native de votre application :

```
user_doc =
    { "nom d'utilisateur":
    "foouser",
        "twitter": { "nom d'utilisateur":
        "footwitter", "mot de
        passe": "secret", "email": "twitter@example.com"
    },
    "facebook":
        { "nom d'utilisateur":
        "foofacebook", "mot de
        passe": "secret", "email": "facebook@example.com"
    },
    "irc":
        { "nom d'utilisateur":
        "fooirc", "mot de passe": "secret",
    }
}
```

Notez que dans les documents MongoDB, tout comme dans les dictionnaires Python, les noms de propriétés (c'est-à-dire les clés) sont uniques. En d'autres termes, un seul document ne peut avoir qu'une seule propriété « nom d'utilisateur ». Cette règle s'applique également aux propriétés des sous-documents incorporés. Cette contrainte d'unicité peut en fait être exploitée et permettre certains modèles utiles. Plus précisément, consultez la section intitulée « Modèle de comptabilité rapide ».

Bien entendu, les sous-documents intégrés peuvent être interrogés, tout comme leurs homologues de niveau supérieur. Par exemple, il serait tout à fait légal de tenter d'interroger le document ci-dessus dans une collection appelée « utilisateurs » avec l'instruction suivante :

```
user_doc = dbh.users.find_one({"facebook.username": "foofacebook"})
```

Comme vous pouvez le voir, le point (« . ») est utilisé pour désigner les clés dans un sous-document intégré. Cela devrait être familier à quiconque a travaillé avec des objets en JavaScript, où l'accès de style objet via la notation par points peut être utilisé en parallèle avec l'accès de style dictionnaire via des crochets. Comme MongoDB utilise énormément JavaScript en interne, ce choix de notation n'est pas surprenant. Après tout, JSON est une notation d'objet JavaScript. La notation par points peut également être utilisée dans les instructions update avec des modificateurs update tels que $set pour définir la valeur d'une sous-propriété individuelle :

```
# Les modificateurs de mise à jour tels que $set prennent également en
charge la notation par points
    dbh.users.update({"facebook.username":"foofacebook"}, {"$set":{"facebook.username":"bar"}}, safe =Vrai)
```

Cette utilisation de sous-documents intégrés est utile, mais peut-être encore plus utile consiste à intégrer plusieurs sous-documents sous une seule clé. Autrement dit, une propriété dont

value est une liste ou un tableau de sous-documents. Dans MongoDB, il s'agit d'une construction légale et très utile. Il s'agit d'une manière très naturelle de modéliser des relations un-à-plusieurs, ou des relations parent-enfant. Prenons l'exemple d'un document « utilisateur » qui peut référencer plusieurs adresses e-mail pour cet utilisateur. Dans le modèle relationnel, cela serait généralement réalisé avec deux tables : une pour les utilisateurs et une pour les adresses e-mail associées à chaque utilisateur. Une requête de jointure pourrait alors être utilisée pour récupérer un utilisateur avec chacune de ses adresses e-mail.

Dans MongoDB, une approche naturelle pour modéliser une relation un-à-plusieurs serait simplement d'avoir une propriété « emails » sur le document utilisateur, dont la valeur est un tableau contenant des sous-documents, chacun représentant un compte de messagerie associé. Par exemple:

```
# Un document utilisateur démontrant les relations un-à-plusieurs grâce à l'intégration.
# Ici, nous mappons plusieurs adresses e-mail (en indiquant si l'e-mail # est ou non
l'adresse e-mail principale de l'utilisateur) à un seul utilisateur.
user_doc =
    { "nom d'utilisateur":
    "foouser",

        "emails":[ { "email":



        "foouser1@example.com", "primary":
          True }, { "email":


        "foouser2@example2.com", "
          primaire":False },
        { "email":"foouser3@example3.com", "primary":False }
    ]
}
```

Non seulement cela fonctionne, mais MongoDB possède des fonctionnalités spécifiques pour vous aider à travailler avec ce type de structure embarquée. Tout comme vous pouvez interroger des documents par la valeur des sous-documents directement incorporés dans le document de niveau supérieur, les documents peuvent également être localisés par la valeur des sous-documents incorporés dans des tableaux. Pour ce faire, utilisez simplement la même notation point (« . ») Comme décrit précédemment dans cette section. MongoDB recherche de manière transparente les sous-documents dans les tableaux.

Revenant à notre exemple précédent d'un seul utilisateur avec plusieurs adresses e-mail, considérons le code suivant :

```
# Un document utilisateur démontrant les relations un-à-plusieurs en utilisant
l'intégration
    user_doc =


    { "username":"foouser", "emails":

        [ { "email":"foouser1@example.com", "primary":True },
```

```
            { "email": "foouser2@example2.com",
              "primary": False },

            { "email": "foouser3@example3.com",
              "primary": False }

        ]
}
# Insérer le document utilisateur
dbh.users.insert(user_doc, safe=True)
# Récupérez le document qui vient d'être inséré via l'une de ses nombreuses adresses
email user_doc_result = dbh.users.find_one({"emails.email":"foouser1@example.com"})
# Affirmer que le document utilisateur d'origine et le résultat de la requête sont identiques
assert user_doc == user_doc_result
```

En plus de MongoDB comprenant les listes de sous-documents pour permettre l'interrogation des valeurs intégrées via la notation par points, il existe également des modificateurs de mise à jour utiles. $pull, $push et leurs variantes sont les plus utiles, permettant l'ajout et la suppression atomiques de sous-documents vers et depuis des listes intégrées. Prenons le cas où un utilisateur ne souhaite plus qu'une adresse e-mail particulière soit liée à son compte. La manière naïve de supprimer cette adresse e-mail de leur document utilisateur serait d'abord de rechercher leur document utilisateur, de le modifier dans le code de votre application afin qu'il ne contienne plus l'adresse e-mail supprimée, puis d'envoyer une requête de mise à jour à la base de données. Non seulement cela est fastidieux, mais cela introduit également une condition de concurrence critique, car le document utilisateur sous-jacent peut avoir été modifié par un autre processus entre votre lecture et votre écriture :

```
# Méthode naïve pour supprimer une adresse e-mail d'un document utilisateur #
Encombrante et présente une condition de
concurrence
    user_doc =

    { "username":"foouser", "emails":

        [ { "email":"foouser1@example.com", "primary
          ":True },

        { "email":"foouser2@example2.com",
          "primary":False },
        { "email":"foouser3@example3.com", "primary":False }
    ]
}
# Insérer le document utilisateur
dbh.users.insert(user_doc, safe=True)
# Récupérez le document qui vient d'être inséré via le nom
d'utilisateur user_doc_result = dbh.users.find_one({"username":"foouser"})
# Supprimez le sous-document d'adresse e-mail "foouser2@example2.com" de la liste intégrée del
user_doc_result["emails"][1]
```

```
# Maintenant, écrivez la nouvelle propriété emails dans la base
de données # Peut entraîner une perte de données en raison de la course entre la
lecture et l'écriture dbh.users.update({"username":"foouser"},{"$set":{"emails ":user_doc_result}},
safe=True)
```

Les trois opérations les plus courantes sur les sous-documents incorporés dans une propriété de liste sont : la suppression, l'insertion et la modification. Chacune de ces opérations peut être effectuée de manière atomique avec les modificateurs de mise à jour fournis. Montrons d'abord l'utilisation de $pull pour supprimer le sous-document correspondant à « foouser2@example2.com » d'une manière simple et sans course :

```
# Supprimez atomiquement une adresse e-mail d'un document utilisateur sans race à l'aide du #
$pull update modificateur
user_doc =
    { "username":"foouser",
    "emails":

        [ { "email":"foouser1@example.com", "
          primaire":Vrai },


        { "email":"foouser2@example2.com",
          "primary":False },


        { "email":"foouser3@example3.com",
          "primary":False }

    ]
}
# Insérer le document utilisateur
dbh.users.insert(user_doc, safe=True)
# Utilisez $pull pour supprimer atomiquement le sous-document de courrier électronique
"foouser2@example2.com"
    dbh.users.update({"username":"foouser"}, {"$pull":{"emails":{"email" :"foouser2@example2.com"}}}, safe=True)
```

Dans cet exemple, $pull est utilisé pour faire correspondre un document intégré avec « email » : « foouser2@example2.com » dans le champ « emails ». $pull supprimera l'intégralité du document du tableau de manière atomique, ce qui signifie qu'il n'y a aucune possibilité de condition de concurrence critique. Vous pouvez également utiliser des modificateurs de requête avec $pull, par exemple pour supprimer tous les sous-documents avec une valeur « primaire » qui n'est pas égale à True, vous pourriez écrire ce qui

```
suit : # Utilisez $pull pour supprimer atomiquement tous les sous-documents de courrier électronique
avec primaire différent de True dbh.users.update({"username":"foouser"},
    {"$pull":{"emails":{"primary":{"$ne":True}}}, safe=True)
```

La gamme complète des modificateurs de requête (voir le tableau du chapitre 2) est disponible, y compris $gt, $lt et ainsi de suite. De plus, $pull peut être utilisé avec des tableaux contenant des atomes (nombres, chaînes, dates, ObjectID, etc.). En d'autres termes, $pull ne fonctionne pas uniquement avec les documents intégrés : si vous stockez une liste de types primitifs dans un tableau, vous pouvez également supprimer des éléments de manière atomique avec $pull .

Le modificateur $push update est utilisé pour ajouter atomiquement un élément à un tableau. Au moment de la rédaction, $push ne peut prendre en charge que l'ajout d'éléments à la fin du tableau : il n'existe pas de modificateur de mise à jour pour ajouter un élément au début d'un tableau ou pour l'insérer à un index arbitraire. $push est simple à utiliser car, contrairement à $pull, il ne prend aucune correspondance de champ ni argument conditionnel.

Par exemple, pour ajouter atomiquement une nouvelle adresse e-mail à notre document utilisateur, nous pourrions utiliser la requête suivante :

```
# Utilisez $push pour ajouter atomiquement un nouveau sous-document de courrier électronique au
document utilisateur new_email = {"email":"fooemail4@exmaple4.com",
"primary":False} dbh.users.update({"username":" foouser"},
    {"$push":{"emails":new_email}}, safe=True)
```

Le dernier cas consiste à mettre à jour un sous-document existant sur place. Ceci peut être réalisé en utilisant ce qu'on appelle l'opérateur « positionnel ». L'opérateur positionnel est représenté par le signe dollar (« $ »). Fondamentalement, il est remplacé par le serveur avec l'index de l'élément correspondant à la spécification du document. Par exemple, supposons que nous souhaitions rendre l'adresse e-mail « foouser2@example2.com » de notre document utilisateur principale. Nous pourrions émettre la requête de mise à jour suivante pour le modifier sur place : # Démontrer l'utilisation de

```
l'opérateur positionnel ($) pour modifier # les sous-documents
correspondants sur place. user_doc =
{ "nom
    d'utilisateur": "foouser",
    "emails":

        [ { "email": "foouser1@example.com",
          "primary": True },


        { "email": "foouser2@example2.com", "
          primaire":False },


        { "email":"foouser3@example3.com",
          "primary":False }

    ]
}
# Insérer le document utilisateur
dbh.users.insert(user_doc, safe=True)
# Maintenant, définissez l'adresse e-mail "foouser2@example2.com" comme
principale dbh.users.update({"emails.email":"foouser2@example2.com"},
    {"$set":{"emails.$.primary" :Vrai}}, sûr=Vrai)
# Maintenant, faites en sorte que l'adresse e-mail "foouser1@example.com" ne
soit pas principale dbh.users.update({"emails.email":"foouser1@example.com"},
    {"$set":{"emails.$.primary ":Faux}}, coffre-fort=Vrai)
```

Notez que l' opérateur $ ne peut pas être utilisé avec les upserts (voir la section sur les upserts plus loin dans ce chapitre). De plus, il ne fonctionne qu'avec le premier élément correspondant.

Lorsque vous travaillez avec l'intégration, il est important d'être conscient des caractéristiques de performances des documents et sous-documents dans MongoDB. Tout d'abord, lorsqu'un document est extrait de la base de données pour répondre à une requête, le document entier, y compris tous les sous-documents intégrés, est chargé en mémoire. Cela signifie qu'il n'y a aucun coût supplémentaire (mis à part le réseau supplémentaire et la surcharge CPU de décodage/codage induite par un ensemble de résultats plus grand) pour récupérer les données intégrées. Une fois le document de niveau supérieur récupéré, ses sous-documents sont également immédiatement disponibles. Comparez cela avec une conception de schéma relationnel utilisant des jointures, où la base de données peut avoir besoin de lire une ou plusieurs tables supplémentaires pour récupérer les données associées. Selon la situation, ces jointures peuvent avoir un impact considérable sur les performances des requêtes.

Deuxièmement, il est également très important de savoir qu'il existe une limite de taille pour les documents dans MongoDB. De plus, la limite de taille des documents a été augmentée au fil des versions majeures successives de MongoDB. Dans MongoDB 1.4.x et 1.6.x, la taille maximale du document était de 4 Mo, mais dans la version 1.8.x, elle a été augmentée à 16 Mo. On peut s'attendre à ce que cette limite continue d'augmenter – peut-être éventuellement jusqu'à devenir arbitrairement grande – mais pour l'instant, gardez à l'esprit que les documents ont une taille finie lors de la modélisation de vos données.

Dans la pratique, il est rare d'atteindre ne serait-ce qu'une taille de document de 4 Mo, à moins que la conception ne soit telle que les documents continuent de s'agrandir au fil du temps. Par exemple, un scénario dans lequel de nouvelles propriétés sont créées sur une base horaire ou quotidienne. Dans de tels cas, il est sage de s'assurer qu'il existe une certaine logique d'application pour gérer la purge des sous-documents intégrés anciens/expirés afin d'éviter que la limite ne soit atteinte.

Un autre exemple serait de créer une plate-forme de publication de documents intégrant chaque document publié par un utilisateur en tant que sous-document à l'intérieur du document utilisateur.
Même si les performances seraient excellentes puisqu'une seule requête sur le document utilisateur pourrait récupérer tous leurs documents publiés en une seule fois, il est fort probable que certains utilisateurs finissent par publier plus de 16 Mo de contenu.

Par conséquent, il faut souvent faire preuve de jugement lors de la conception de schémas MongoDB : intégrer ou ne pas intégrer.

L'alternative à l'intégration consiste à stocker les documents dans une collection distincte et à effectuer une jointure dans votre propre code d'application, en interrogeant deux fois ou plus. Habituellement, les relations plusieurs-à-plusieurs sont modélisées de cette manière, tandis que les relations un-à-plusieurs sont intégrées.

Recherches rapides : utilisation d'index avec MongoDB Le rôle des index dans

MongoDB est très similaire à leur rôle dans les SGBDR traditionnels tels que MySQL, PostgreSQL, etc. MongoDB propose deux types d'index prêts à l'emploi : les index Btree et les index géospatiaux. Les index btree dans MongoDB sont sensiblement les mêmes que leurs équivalents dans MySQL ou PostgreSQL. Lorsque, dans un système relationnel, vous placez un index sur une colonne pour obtenir des recherches rapides sur ce champ, vous faites une chose analogue dans MongoDB en plaçant un index sur une propriété particulière dans une collection. Tout comme

avec un SGBDR, les index MongoDB peuvent s'étendre sur plusieurs champs (appelés index composés), ce qui est utile si vous savez à l'avance que vous effectuerez une requête en fonction de la valeur de plusieurs propriétés. Un index composé serait utile par exemple si vous interrogez des documents par prénom et nom. Dans MongoDB, les index btree peuvent avoir une « direction ». Cette direction n'est utile que dans le cas d'index composés, où la direction de l'index doit correspondre à la direction du tri ou à la direction de la requête de plage pour des performances optimales. Par exemple, si vous interrogez une plage (par exemple, A à C) sur le prénom et le nom, puis que vous triez par ordre croissant sur le nom de famille, la direction de votre index composé doit également être ascendante.

L'utilisation d'un index btree entraînera une baisse des performances lors des écritures, car la base de données doit désormais mettre à jour l'index en plus des données. Pour cette raison, il est judicieux de choisir soigneusement vos index. Évitez autant que possible les index superflus. Les index occupent également un espace de stockage précieux (ce qui n'est pas vraiment un problème avec l'espace disque aujourd'hui étant donné le faible prix par téraoctet), mais aussi en mémoire. Votre base de données fonctionnera plus rapidement lorsqu'elle réside entièrement en mémoire, et les index peuvent considérablement augmenter sa taille. Il s'agit d'un scénario classique de compromis entre le temps et l'espace en informatique.

Les index btree MongoDB peuvent également être utilisés pour appliquer une contrainte unique sur une propriété particulière dans une collection. Par défaut, la propriété de clé primaire _id possède un index unique créé dans MongoDB. La contrainte unique empêchera la propriété protégée d'avoir une valeur en double au sein de la collection. Cela peut être utile pour les valeurs qui devraient être globalement uniques dans la collection, un exemple courant étant les noms d'utilisateur. Attention cependant à une dépendance excessive à l'égard de cette fonctionnalité, car dans l'implémentation actuelle du partitionnement, les index uniques ne sont pris en charge que sur la propriété _id , sinon ils ne sont pas appliqués globalement à travers le cluster.

Les index Btree prennent également en charge de manière transparente l'indexation des propriétés à valeurs multiples, c'est-à-dire les propriétés dont la valeur est un tableau. Chaque élément du tableau sera correctement stocké dans l'index pour permettre une récupération rapide du document parent. Cela peut être utile pour les implémentations performantes de balisage, où chaque balise est stockée sous forme de chaîne dans une propriété de liste « tags » sur le document. Les recherches de documents correspondant à une ou plusieurs de ces balises (utilisant potentiellement l' opérateur de requête $in) seront ensuite recherchées dans l'index « tags ».
De plus, les index btree sont également bien pris en charge lorsqu'ils sont placés sur des sous-documents intégrés. Si, par exemple, vous stockez des adresses e-mail sous forme de sous-documents intégrés et que vous souhaitez pouvoir rechercher la valeur de l'adresse e-mail à l'aide d'un index, MongoDB le permet. Par conséquent, le document et la requête suivants pourraient tirer parti d'un index :

```
user_doc =
    { "nom d'utilisateur":
    "foouser",

        "emails":[ { "email":


        "foouser1@example.com", "primary": True }, { "email": "foouser2@example2.com",
```

```
                "primary":False },


              { "email":"foouser3@example3.com",
                "primary":False }

        ]
}

dbh.users.insert(user_doc)
# Si nous plaçons un index sur la propriété "emails.email", # par
exemple dbh.users.create_index("emails.email") # cette
requête find_one peut utiliser un index btree user =
dbh.users.find_one({"emails.email ":"foouser2@exemple2.com"})
```

Les index Btree dans MongoDB sont également importants lors du tri des résultats côté serveur. Sans index sur la propriété sur laquelle vous effectuez le tri, MongoDB manquera de mémoire lorsque vous tenterez de trier quelque chose de plus grand qu'un ensemble de résultats relativement petit (environ 4 Mo au moment de la rédaction). Si vous prévoyez de trier des ensembles de résultats supérieurs à 4 Mo, vous devez spécifier un index sur la clé de tri. Il est facile de sous-estimer cela et de constater que des exceptions sont générées lors de requêtes portant sur des données réelles plus volumineuses, ce qui n'était pas prévu lors du développement.

Pour créer un index avec le pilote PyMongo, utilisez la méthode Collection.create_index() . Cette méthode peut créer des index à clé unique ou des index composés. Pour un index à clé unique, seule la clé doit être fournie. Un index composé est légèrement plus compliqué : une liste de 2 tuples (clé, direction) doit être fournie.

Par exemple, pour créer un index sur la propriété username d'une collection appelée users, vous pouvez écrire ce qui suit :

```
# Créer un index sur la propriété du nom
d'utilisateur dbh.users.create_index("username")
```

Pour créer un index composé, par exemple sur le prénom et le nom, avec un sens ascendant, vous pouvez spécifier :

```
# Créer un index composé sur les propriétés prénom
et nom # avec un sens d'index croissant dbh.users.create_index([("first_name",
pymongo.ASCENDING), ("nom",
pymongo.ASCENDING)])
```

Les index dans MongoDB ont chacun des noms. Par défaut, MongoDB générera un nom, mais vous souhaiterez peut-être donner un nom personnalisé, en particulier pour les index composés où les noms générés ne sont pas particulièrement lisibles par les humains. Pour donner un nom personnalisé lors de la création, fournissez le paramètre name=<str> à la méthode create_index() :

```
# Créez un index composé appelé "name_idx" sur les propriétés first_name et last_name # avec la direction
d'index ascendante

dbh.users.create_index([ ("first_name",
      pymongo.ASCENDING), ("last_name",

      pymongo.ASCENDING) ], name=" nom_idx")
```

A noter que la création d'index verrouille la base de données par défaut. Pour les grandes collections, la création d'un index peut prendre du temps. Pour aider à atténuer l'impact de ces opérations sur les bases de données de production en direct, MongoDB est capable de créer des index en arrière-plan, sans bloquer l'accès aux bases de données. La création d'un index en arrière-plan peut prendre un peu plus de temps et entraînera toujours une charge supplémentaire sur le système, mais la base de données devrait autrement rester disponible.

Pour spécifier qu'un index doit être construit en arrière-plan, passez le paramètre background=True à la méthode create_index() :

```
# Créer un index en arrière-plan # La base de
données reste utilisable
dbh.users.create_index("username", background=True)
```

Comme mentionné précédemment dans cette section, les index btree MongoDB peuvent être utilisés pour appliquer une contrainte d'unicité sur une propriété particulière. Des contraintes uniques peuvent être appliquées à la fois aux index à clé unique et aux index composés. Pour créer un index avec une contrainte d'unicité, passez simplement le paramètre unique=True à la méthode create_index() :

```
# Créer un index avec une contrainte unique sur la propriété du nom
d'utilisateur dbh.users.create_index("username", unique=True)
```

Sachez que les index uniques dans MongoDB ne fonctionnent pas exactement de la même manière que les index dans les systèmes SGBDR. En particulier, un document avec une propriété manquante sera ajouté à l'index comme si la valeur de cette propriété était nulle. Cela signifie que lorsqu'une contrainte unique est ajoutée à un index btree dans MongoDB, la base de données vous empêchera d'avoir plusieurs documents dans la collection pour lesquels il manque la propriété indexée.
Par exemple, si vous avez créé un index unique pour la propriété de nom d'utilisateur dans une collection d'utilisateurs, un seul document de cette collection peut être autorisé à ne pas posséder de propriété de nom d'utilisateur . Les écritures de documents supplémentaires sans propriété de nom d'utilisateur déclencheront une exception. Si vous essayez d'ajouter un index unique à une collection qui contient déjà des doublons sur la propriété spécifiée, MongoDB lèvera (sans surprise) une exception.
Cependant, si cela ne vous dérange pas de supprimer les données en double, vous pouvez demander à MongoDB de supprimer tous les documents trouvés sauf le premier à l'aide du paramètre dropDups ou drop_dups :

```
# Créer un index avec une contrainte unique sur la propriété du nom
d'utilisateur # demander à MongoDB de supprimer tous les doublons après le premier document
trouvé. dbh.users.create_index("nom d'utilisateur", unique=True, drop_dups=True)
# Pourrait également s'écrire : #
dbh.users.create_index("username", unique=True, dropDups=True)
```

Au fil du temps, votre schéma peut évoluer et vous constaterez peut-être qu'un index particulier n'est plus nécessaire. Heureusement, les index sont faciles à supprimer dans MongoDB. La méthode Collection.drop_index() supprime un index à la fois. Si vous avez créé votre index avec un nom personnalisé (comme décrit ci-dessus), vous devez fournir ce même nom à la méthode drop_index() afin de le supprimer. Par exemple:

```
# Créer un index sur la propriété du nom d'utilisateur appelé
"username_idx" dbh.users.create_index("username", name="username_idx")
# Supprimer l'index appelé "username_idx"
dbh.users.drop_index("username_idx")
```

Si, en revanche, vous n'avez pas donné de nom personnalisé à votre index, vous pouvez le supprimer en transmettant le spécificateur d'index d'origine. Par exemple:

```
# Créez un index composé sur les propriétés first_name et last_name # avec une direction
d'index ascendante
dbh.users.create_index([("first_name", pymongo.ASCENDING), ("last_name", pymongo.ASCENDING)])

# Supprimer cet index
dbh.users.drop_index([("prénom", pymongo.ASCENDING), ("nom", pymongo.ASCENDING)])
```

Tous les index d'une collection peuvent être supprimés en une seule instruction à l'aide de la méthode Collection.drop_indexes() .

Si vous souhaitez inspecter par programme les index de vos collections à partir de Python, vous pouvez utiliser la méthode Collection.index_information() . Cela renvoie un dictionnaire dans lequel chaque clé est le nom d'un index. La valeur associée à chaque clé est un dictionnaire supplémentaire. Ces dictionnaires de deuxième niveau contiennent toujours une clé spéciale appelée clé, qui est une entrée contenant le spécificateur d'index d'origine, y compris la direction de l'index. Ce spécificateur d'index d'origine correspondait aux données transmises à la méthode create_index() lors de la première création de l'index. Les dictionnaires de deuxième niveau peuvent également contenir des options supplémentaires telles que des contraintes uniques, etc.

Applications basées sur la localisation avec MongoDB : indexation géospatiale

Comme mentionné dans la section précédente sur les index, MongoDB prend en charge deux types d'index : Btree et géospatial. Les index Btree ont été traités de manière assez approfondie dans la section précédente, mais nous n'avons pas encore décrit les index GeoSpatial.

Tout d'abord, voyons pourquoi l'indexation géospatiale pourrait être utile. Aujourd'hui, de nombreuses applications sont créées avec l'exigence d'une connaissance de la localisation. Généralement, cela se traduit par des fonctionnalités dans lesquelles les points d'intérêt (POI) proches d'un emplacement utilisateur particulier peuvent être rapidement récupérés à partir d'une base de données. Par exemple, une application mobile géolocalisée pourrait souhaiter récupérer rapidement une liste des coffeeshops à proximité, en fonction des coordonnées GPS actuelles. Le problème, fondamentalement, est que le monde est à la fois assez vaste et assez rempli de points intéressants - et donc essayer de répondre à une telle requête en parcourant la liste complète de tous les POI du monde pour trouver ceux qui sont à proximité prendrait une période inacceptablement longue. D'où la nécessité d'une sorte d'indexation GeoSpatial, pour accélérer ces recherches.

Heureusement pour toute personne chargée de créer des applications géolocalisées, MongoDB est l'une des rares bases de données à prendre en charge l'indexation géospatiale dès le départ.
MongoDB utilise le géohashing, un algorithme du domaine public développé par Gustavo Nie-meyer, qui traduit la proximité géographique en proximité lexicale. Par conséquent, une base de données prenant en charge les requêtes de plage (telle que MongoDB) peut être utilisée efficacement pour rechercher des points proches et dans les limites.

Il convient de noter qu'à l'heure actuelle, la prise en charge de l'indexation géospatiale de MongoDB se limite uniquement aux requêtes basées sur des points. Les opérateurs fournis ne peuvent être utilisés que pour rechercher des points individuels, et non des itinéraires ou des sous-formes.

MongoDB fournit les opérateurs $near et $within qui constituent le principal moyen d'effectuer des requêtes géospatiales dans le système. En utilisant $near, vous pouvez trier efficacement les documents d'une collection en fonction de leur proximité avec un point donné. L' opérateur $within vous permet de spécifier des limites pour la requête. Les définitions de limites prises en charge incluent $box pour une forme rectangulaire et $circle pour un cercle. Dans MongoDB 1.9 et versions ultérieures, l' opérateur $poly gon permet des limites de polygones convexes et concaves.

Avant de pouvoir utiliser les requêtes géospatiales, vous devez disposer d'un index géospatial. Dans les versions Mon-goDB jusqu'à 1.8.x incluse, les index géospatiaux sont limités à un seul index par collection. Cela signifie que chaque document ne peut avoir qu'une seule propriété d'emplacement interrogée efficacement par MongoDB. Cela peut avoir des implications importantes pour la conception de schémas, c'est pourquoi il est bon de le savoir dès le départ.

Les index géospatiaux limitent par défaut les valeurs acceptables pour la propriété de localisation sur les documents à celles du GPS. Autrement dit, les coordonnées doivent être comprises entre -180 .. +180. Si vous avez des coordonnées en dehors de cette plage, MongoDB déclenchera une exception lorsque vous tenterez de créer l'index géospatial sur la collection. Si vous souhaitez indexer des valeurs en dehors de la plage du GPS standard, vous pouvez le spécifier au moment de la création de l'index.

La propriété location sur vos documents doit être soit un tableau, soit un sous-document où les deux premiers éléments sont les coordonnées x et y à indexer. L'ordre des coordonnées (que ce soit x,y ou y,x) n'a pas d'importance tant qu'il est cohérent sur tous les documents. Par exemple, votre document pourrait ressembler à l'un des éléments suivants :

```
# la propriété location est un tableau avec x,y ordonnant
user_doc =
      { "username":"foouser",
      "user_location":[x,y]
}

# la propriété location est un tableau avec y,x ordonnant
user_doc =
      { "username":"foouser",
      "user_location":[y,x]
}

import bson
# location property est un sous-document avec l'ordre y,x loc =
bson.SON()
loc["y"] = y
loc["x"] = x
user_doc =
      { "username":"foouser",
      " emplacement_utilisateur":loc
}
```

```
import bson
# location property est un sous-document avec l'ordre x,y loc =
bson.SON() loc["x"]
= x loc["y"] = y
user_doc =

{ "username":"foouser", "
      emplacement_utilisateur":loc
}
```

Notez qu'en Python, le type de dictionnaire par défaut (classe dict), l'ordre n'est pas conservé. Lorsque vous utilisez location dans un sous-document de Python, utilisez plutôt un objet bson.SON . bson.SON est livré avec le pilote PyMongo et est utilisé exactement de la même manière que la classe dict de Python .

Une fois que les propriétés de localisation des documents de votre collection sont correctement formées, nous pouvons créer l'index géospatial. Comme pour les index btree, les index géospatiaux dans Mon-goDB sont créés avec la méthode Collection.create_index() de PyMongo . En raison de la limitation d'un index géospatial par collection dans les versions de MongoDB jusqu'à et y compris 1.8.x, si vous prévoyez d'effectuer une requête par d'autres propriétés en plus de la propriété d'emplacement, vous pouvez faire de votre index géospatial un composé indice. Par exemple, si vous savez que vous effectuerez une recherche dans votre collection à la fois par les propriétés « nom d'utilisateur » et « emplacement_utilisateur », vous pouvez créer un index géographique composé dans les deux champs. Cela peut aider à contourner la limitation de l'index géospatial unique dans de nombreux cas.

En revenant à notre exemple de documents dans une collection appelée « utilisateurs » avec la propriété location étant « user_location », nous créerions un index géospatial avec le suivant déclaration:

```
# Créez un index géospatial sur la propriété "user_location".
dbh.users.create_index([("emplacement_utilisateur", pymongo.GEO2D)])
```

Pour créer un index géospatial composé qui nous permettrait d'interroger efficacement l'emplacement et le nom d'utilisateur, nous pourrions émettre cette déclaration :

```
# Créez un index géospatial sur la propriété "user_location".
dbh.users.create_index([("emplacement_utilisateur", pymongo.GEO2D), ("nom
d'utilisateur", pymongo.ASCENDING)])
```

Maintenant que nous disposons d'index géospatiaux, nous pouvons essayer des requêtes efficaces basées sur la localisation. L' opérateur $near est assez facile à comprendre, nous allons donc commencer par là. Comme cela a déjà été expliqué, $near triera les résultats de la requête par proximité du point spécifié. Par défaut, $near tentera de trouver les 100 résultats les plus proches.

Une considération importante en matière de performances qui n'est pas clairement mentionnée dans la documentation officielle de MongoDB est que lorsque vous utilisez $near, vous souhaiterez presque toujours spécifier une distance maximale sur la requête. Sans limitation de la distance maximale, afin de renvoyer le nombre de résultats spécifié (par défaut 100), MongoDB doit effectuer une recherche dans toute la base de données. Cela prend beaucoup de temps. Dans la plupart des cas, une distance maximale de

environ 5 degrés devraient suffire. Puisque nous utilisons des coordonnées en degrés décimaux (c'est-à-dire GPS), l'unité de distance maximale est le degré. 1 degré équivaut à environ 69 miles. Si vous ne vous souciez que d'un ensemble de résultats relativement restreint (par exemple, les 10 cafés les plus proches), limiter la requête à 10 résultats devrait également améliorer les performances.

Commençons par un exemple de recherche des 10 utilisateurs les plus proches du point 40, 40 en limitant une distance maximale de 5 degrés :

```
# Trouvez les 10 utilisateurs les plus proches du point 40, 40 avec une distance maximale
de 5 degrés close_users =

    dbh.users.find( {"user_location":
        {"$near" : [40, 40], "$maxDistance":5}} ).limite(10)
# Imprime les utilisateurs
pour l'utilisateur dans les utilisateurs
    les plus proches : # suppose que la propriété user_location est
    un tableau x,y print "L'utilisateur %s est à l'emplacement %s,%s" %(user["username"], user["user_location"][0] ,
        utilisateur["emplacement_utilisateur"[1])
```

Essayons ensuite d'utiliser l' opérateur géospatial $within pour trouver des points à l'intérieur d'une certaine limite. Cela peut être utile lors de la recherche de POI dans un comté/une ville spécifique ou même dans un quartier bien défini au sein d'une ville. Dans le monde réel, ces frontières sont floues et changent constamment, mais il existe des bases de données suffisamment performantes pour qu'elles soient utiles.

Pour spécifier un rectangle dans lequel effectuer la recherche, vous fournissez simplement les coordonnées inférieure gauche et supérieure droite sous forme d'éléments dans un tableau. Par exemple:

```
boîte = [[50.73083, -83.99756], [50.741404, -83.988135]]
```

Nous pourrions rechercher des points à l'intérieur de cette limite en utilisant la requête géospatiale

```
suivante : box = [[50.73083, -83.99756], [50.741404, -83.988135]]
users_in_boundary = dbh.users.find({"user_location":{"$within" : {"$boîte":boîte}}})
```

Pour spécifier un centre de recherche dans lequel effectuer la recherche, il vous suffit de fournir le point central et le rayon. Comme pour $maxDistance mentionné précédemment, les unités du rayon sont les degrés. Voici comment nous pourrions effectuer une recherche géospatiale pour 10 utilisateurs dans un rayon de 5 degrés centré au point 40, 40 :

```
users_in_circle = dbh.users.find({"user_location":{"$within":{"$center":[40, 40, 5]}}}).limit(10)
```

Notez qu'avec la limite du cercle utilisant $center, nous passons un tableau dont les deux premières valeurs sont les coordonnées du centre et le troisième paramètre est le rayon (en degrés).

Toutes les requêtes que nous avons mentionnées jusqu'à présent et qui utilisent un index géospatial ne sont en réalité pas entièrement exactes. En effet, ils utilisent un modèle de terre plate dans lequel chaque degré d'arc de latitude et de longitude se traduit par la même distance partout sur la terre.
En réalité, la terre est une sphère et ces valeurs diffèrent donc selon l'endroit où l'on se trouve. Heureusement, MongoDB dans les versions 1.8.x et supérieures implémente un modèle sphérique de la Terre pour les requêtes géospatiales.

Le nouveau modèle sphérique peut être utilisé en utilisant les variantes $nearSphere et $circle Sphere sur les opérateurs $near et $circle . Le modèle sphérique de MongoDB comporte quelques mises en garde supplémentaires. Avant toute chose, vous devez utiliser l'ordre (longitude, latitude) de vos coordonnées. Bien qu'il existe de nombreuses autres applications et formats qui utilisent l'ordre (latitude, longitude), vous devez faire attention à réorganiser pour l'utiliser avec le modèle sphérique de MongoDB. Deuxièmement, contrairement aux opérateurs $near et $center que nous venons de décrire, les unités de distance avec $nearSphere et $centerSphere sont toujours exprimées en radians. Cela inclut l'utilisation de $maxDistance avec $nearSphere ou $centerSphere.

Heureusement, il n'est pas difficile de convertir une unité plus compréhensible par l'homme, comme le kilomètre, en radians. Pour traduire des kilomètres en radians, divisez simplement la valeur du kilomètre par le rayon de la terre qui est d'environ 6 371 km (ou 3 959 miles). Pour le démontrer, essayons notre exemple précédent consistant à trouver les 10 utilisateurs les plus proches du point 40,40 avec une distance maximale de 5 km, mais cette fois en utilisant le modèle sphérique :

```
# Trouvez les 10 utilisateurs les plus proches du point 40, 40 avec une distance maximale de 5 degrés #
Utilise le modèle sphérique fourni par MongoDB 1.8.x et versions ultérieures

earth_radius_km = 6371.0
max_distance_km = 5.0
max_distance_radians = max_distance_km / earth_radius_km close_users =
dbh.users.find( {"user_location":
    {"$nearSphere" : [40,
        40],
          "$maxDistance":max_distance_radians}}).limit(10)
# Imprime les utilisateurs
pour l'utilisateur dans les utilisateurs
    les plus proches : # suppose que la propriété user_location est
    un tableau x,y print "L'utilisateur %s est à l'emplacement %s,%s" %(user["username"], user["user_location"][0] ,
        utilisateur["emplacement_utilisateur"[1])
```

Codez de manière défensive pour éviter les erreurs de clé et autres bugs

L'un des inconvénients d'une base de données orientée document par rapport à une base de données relationnelle est que la base de données n'applique pas de schéma à votre place. Pour cette raison, lorsque vous travaillez avec MongoDB, vous devez être vigilant dans votre gestion des résultats de la base de données. Ne présumez pas aveuglément que les résultats auront toujours les propriétés que vous attendez.

Vérifiez leur présence avant d'y accéder. Bien que Python génère généralement une KeyError et arrête l'exécution, en fonction de votre application, cela peut toujours entraîner une perte d'intégrité des données. Prenons le cas de la mise à jour de chaque document d'une collection un par un : une seule KeyError imprévue pourrait laisser la base de données dans un état incohérent, certains documents ayant été mis à jour et d'autres non.

Par exemple,

```
all_user_emails = [] pour
le nom d'utilisateur dans ("jill", "sam", "cathy") :
    user_doc = dbh.users.find_one({"username":username})
    # KeyError sera déclenché si l'un de ces éléments n'existe pas
    dbh.emails.insert({"email":user_doc["email"]})
```

Parfois, vous souhaiterez avoir une solution de secours par défaut pour éviter les KeyErrors si un document renvoyé ne contient pas une propriété requise par votre programme. La méthode get de la classe Python dict vous permet facilement de spécifier une valeur par défaut pour une propriété si elle est manquante.

Il est souvent judicieux de l'utiliser de manière défensive. Par exemple, imaginez que nous ayons une collection de documents utilisateur. Certains utilisateurs ont une propriété « score » définie sur un nombre, tandis que d'autres ne l'ont pas. Il serait prudent d'avoir une solution de repli par défaut de zéro (0) dans le cas où aucune propriété de score n'est présente. Nous pouvons considérer qu'un score manquant signifie un score nul. La méthode get de la classe dict nous permet de le faire facilement :

```
total_score = 0 pour
le nom d'utilisateur dans ("jill", "sam", "cathy") : user_doc =
    dbh.users.find_one({"username":username}) total_score +=
    user_doc.get("score", 0)
```

Cette approche peut également bien fonctionner lors d'une boucle sur des listes intégrées. Par exemple, pour gérer de manière défensive le cas où un document représentant un produit particulier n'a pas encore de liste de fournisseurs intégrée (peut-être parce qu'il n'est pas encore sur le marché ou n'est plus produit), vous pouvez écrire du code comme celui-ci :

```
# Envoyez un e-mail à chaque fournisseur de ce produit.
# La valeur par défaut est la liste vide donc aucune casse # spéciale n'est
nécessaire si la propriété des fournisseurs n'est pas présente. pour le
fournisseur dans product_doc.get("fournisseurs", []) :
    email_supplier(fournisseur)
```

MongoDB ne donne également aucune garantie quant au type de valeur d'une propriété sur un document donné.

Dans la plupart des implémentations de SGBDR (l'exception notable que je connais étant SQLite), la base de données appliquera de manière assez rigoureuse les types de colonnes. Si vous essayez d'insérer une chaîne dans une colonne entière, la base de données rejettera l'écriture.

MongoDB, en revanche, ne rejettera de telles écritures que dans des circonstances exceptionnelles.
Si vous définissez la valeur d'une propriété sur un document comme étant une chaîne et que sur un autre document de la même collection, définissez la valeur de cette propriété sur un nombre, il la stockera très volontiers.

```
# Insertion parfaitement légale - MongoDB ne se plaindra pas
dbh.users.insert({"username":"janedoe"})
# Également parfaitement légal - MongoDB ne se plaindra pas
dbh.users.insert({"username":l337})
```

Lorsque vous associez cela à la volonté de Python de vous permettre de renoncer à taper explicitement vos variables, vous pouvez bientôt rencontrer des problèmes. Le scénario le plus courant est peut-être celui de l'écriture des entrées d'applications Web dans la base de données. La plupart des frameworks Python basés sur WSGI vous enverront toutes les valeurs des paramètres HTTP POST et GET sous forme de chaînes, qu'il s'agisse ou non de chaînes.

Ainsi, il est facile d'insérer ou de mettre à jour une propriété numérique avec une valeur qui est une chaîne. Bien entendu, le meilleur moyen d'éviter des erreurs de cette nature est d'empêcher le mauvais type de données

jamais été écrit dans la base de données en premier lieu. Ainsi, dans le contexte d'une application Web, il est fortement conseillé de valider et/ou de contraindre les types de toutes entrées à écrire des requêtes avant de les émettre. Vous pouvez envisager d'utiliser les bibliothèques FormEncode ou Colander Python pour vous aider dans cette validation.

Mise à jour ou insertion : upserts dans MongoDB

Une tâche relativement courante dans une application basée sur une base de données consiste à mettre à jour une entrée existante ou, si elle n'est pas trouvée, à l'insérer en tant que nouvel enregistrement. MongoDB prend en charge cela de manière pratique en une seule opération, vous évitant ainsi d'avoir à implémenter votre propre logique « if-exists-update-else-insert ». 10gen fait référence à ce type d'opération d'écriture comme un « upsert ».

Dans PyMongo, il existe trois méthodes possibles pour effectuer une upsert. Il s'agit de Collection.save(), Collection.update() et Collection.find_and_modify(). Nous commencerons par décrire Collection.save() car c'est la méthode la plus simple.

Dans la section précédente « Insérer un document dans une collection », nous avons utilisé la méthode Collection.insert() pour écrire un nouveau document dans la collection. Cependant, nous aurions tout aussi bien pu utiliser la méthode save() . save() offre des fonctionnalités presque identiques à insert() avec les exceptions suivantes : save() peut effectuer des upserts et save() ne peut pas insérer plusieurs documents en un seul appel. save() est assez facile à comprendre. Si vous lui transmettez

un document sans propriété _id , il effectuera un insert(), créant ainsi un tout nouveau document. Si, par contre, vous lui transmettez un document contenant une propriété _id , il mettra à jour le document existant correspondant à cette valeur _id , en écrasant le document déjà présent par le document passé à save().

C'est l'essence même d'un upsert : si un document existe déjà, mettez-le à jour. Sinon, créez un nouveau document.

save() peut être utile car il prend en charge à la fois l'écriture de nouveaux documents et la modification de documents existants, très probablement ceux récupérés de MongoDB via une requête de lecture. Avoir une seule méthode capable des deux modes de fonctionnement réduit le besoin de gestion conditionnelle dans votre code client, simplifiant ainsi votre programme.

Plus utile, peut-être, est le paramètre « upsert=True » qui peut être transmis à Collection.update(). Comme cela a été discuté dans la section « Mise à jour des documents dans une collection » et décrit plus en détail dans la section « Modificateurs de mise à jour MongoDB », la méthode update() prend en charge l'utilisation de « modificateurs de mise à jour ». Ces opérateurs riches vous permettent d'effectuer des écritures plus complexes que la sémantique de base « écraser mais conserver l'_id existant » de la méthode save() .

Par exemple, imaginez que vous écrivez une méthode, edit_or_add_session(). Cette méthode modifie un document existant ou en insère un nouveau. De plus, la sémantique de la méthode impose que la méthode soit toujours appelée avec un session_id, mais que le session_id puisse ou non être déjà présent dans la base de données. La mise en œuvre naïve-

Il s'agirait d'abord de demander si un document de session existe déjà, puis, sous condition, soit d'insérer un nouveau document de session, soit de mettre à jour le document existant :

```
# Naïf, mauvaise implémentation sans upsert=True
def edit_or_add_session(description, session_id): #
    Nous devons d'abord interroger, car nous ne savons pas si ce session_id existe déjà.
    # Si nous tentons de mettre à jour un document inexistant, aucune écriture n'aura
    lieu. session_doc = dbh.sessions.find_one({"session_id":session_id})
    si session_doc :
        dbh.sessions.update({"session_id":session_id},
            {"$set":{"session_description":description}}, safe= Vrai)
    sinon :
        dbh.sessions.insert({"session_description":description,
"session_id":session_id},
            safe=True)
```

Cependant, en employant la fonctionnalité upsert de Collection.update(), cela peut être implémenté dans un seul appel de méthode, simplifiant considérablement le code et éliminant le besoin d'une requête de lecture supplémentaire :

```
# Bonne implémentation en utilisant
upsert=True def edit_or_add_session(description,
    session_id): dbh.sessions.update({"session_id":session_id},
        {"$set":{"session_description":description}}, safe=True, upsert=True)
```

Notez que nous n'aurions pas pu implémenter la sémantique ci-dessus en utilisant Cursor.save() car nous testons l'existence de la propriété « session_id » plutôt que « _id ».
Rappelons que la méthode upsert save() ne fonctionne qu'avec « _id ».

L'astuce pour comprendre les upserts avec la méthode update() est de considérer les deux cas d'exécution séparément. Dans le cas où le document existe déjà, alors le document de mise à jour sera traité normalement, tout comme avec un appel update() régulier sans le paramètre « upsert=True ». Cependant, dans le cas où le document n'existe pas déjà, le document écrit (inséré) correspondra à la fois à la spécification du document fournie comme premier argument de l' appel update() et au document de mise à jour avec tous les modificateurs qu'il contient. En d'autres termes, le comportement observé est que le document est d'abord créé avec les propriétés spécifiées dans la spécification du document (dans ce cas, "session_id": ses sion_id), puis le document de mise à jour est exécuté en conséquence. Cela ne reflète peut-être pas avec précision ce qui se passe en interne dans le démon ou le pilote, mais cela équivaut à tout ce qui se passe.

Lecture-écriture-modification atomique : findAndModify de MongoDB

Nous avons déjà présenté les modificateurs de mise à jour atomique pris en charge par MongoDB. Ceux-ci sont très puissants et permettent des opérations d'écriture sans course de toutes sortes, y compris la manipulation de tableaux et l'incrémentation/décrémentation. Cependant, il est souvent nécessaire de pouvoir modifier le document de manière atomique, et également de renvoyer le résultat de l'opération atomique, en une seule étape.

Par exemple, imaginez un système de facturation. Chaque document utilisateur possède une propriété « account_balance ». Il peut y avoir des écritures qui modifient le solde du compte, par exemple un événement de recharge du compte qui ajoute de l'argent au compte et une action d'achat qui retire de l'argent du compte. Considérons l'implémentation suivante : # L'utilisateur X ajoute 20 $ à son compte, nous incrémentons

```
donc atomiquement # account_balance dbh.users.update({"username":username},
{"$inc":
{"account_balance":20}} , sûr=Vrai)
# Récupère le solde du compte mis à jour à afficher à l'utilisateur
new_account_balance = dbh.users.find_one({"username":username},
    {"account_balance":1})["account_balance"]
```

Cela fonctionnera correctement en supposant qu'aucune autre écriture sur le solde du compte ne se produise entre les opérations d'écriture et de lecture. Il existe une condition de concurrence évidente. Si une action d'achat devait avoir lieu entre la mise à jour du solde et la lecture du solde, l'utilisateur pourrait être mécontent de se voir présenter un solde après paiement inférieur à ce à quoi il s'attendait !

```
# L'utilisateur X ajoute 20 $ à son compte, nous incrémentons donc
atomiquement #
account_balance dbh.users.update({"username":username}, {"$inc":{"account_balance":20}}, safe=True)

# Entre-temps, dans un autre thread ou processus, il y a une opération de paiement, # qui décrémente le solde du
compte : dbh.users.update({"username":username}, {"$dec":

{"account_balance":5 }}, sûr=Vrai)

# Récupère le solde du compte mis à jour à afficher à l'utilisateur
new_account_balance = dbh.users.find_one({"username":username},
    {"account_balance":1})["account_balance"]
```

Ce que vous voulez dans ce genre de situation, c'est un moyen de mettre à jour le solde du compte et de renvoyer la nouvelle valeur en une seule opération atomique. La commande findAndModify de MongoDB vous permet de faire exactement cela. PyMongo fournit un wrapper autour de findAndModify dans la méthode Collection.find_and_modify() . En utilisant cette méthode, nous pouvons réécrire le code en une seule opération atomique : # L'utilisateur X ajoute 20 $ à son compte, nous incrémentons donc

```
atomiquement # account_balance et renvoyons le document résultant ret =
dbh.users.find_and_modify({"username" :nom d'utilisateur},

    {"$inc":{"account_balance":20}}, safe=True, new=True)
new_account_balance = ret["account_balance"]
```

Modèle de comptabilité rapide

La plupart des applications que les gens créent aujourd'hui sont en temps réel et utilisent de très grands ensembles de données. Autrement dit, les utilisateurs s'attendent à ce que les modifications qu'ils apportent soient reflétées instantanément dans l'application. Par exemple, si un utilisateur remporte un nouveau meilleur score dans un jeu multijoueur, il s'attend à ce que le tableau des meilleurs scores du jeu soit immédiatement mis à jour. Cependant, il se peut qu'il ne s'agisse pas d'un seul tableau des meilleurs scores qui doive être mis à jour. Peut-être êtes-vous également classé par score élevé cette semaine, ou ce mois-ci, ou même cette année. De plus, en tant que développeur d'applications, vous souhaiterez peut-être conserver un journal détaillé de chaque modification, y compris le moment où elle s'est

quelle était l'adresse IP du client, la version du logiciel du client, etc., par utilisateur à des fins d'analyse.

Cette tendance ne se limite pas aux scores élevés. Des exigences comptables de haute performance similaires existent pour les flux d'activités sociales intégrés aux applications, les systèmes de facturation qui facturent par octet, etc. Non seulement ces décomptes doivent être rapides à lire à partir de la base de données, mais ils doivent également être rapides à écrire. De plus, avec potentiellement des millions d'utilisateurs, l'ensemble de données peut devenir très volumineux et très rapidement.

Vous pourriez être tenté de ne conserver qu'un journal détaillé, avec un document par modification.
Les totaux pour les différentes périodes peuvent ensuite être calculés par une requête globale dans la collection. Cela peut bien fonctionner au début, avec seulement des centaines ou des milliers de documents à agréger pour calculer le résultat. Cependant, lorsque le nombre de ces documents atteint des millions, voire des milliards – ce qui peut facilement se produire dans le cadre d'une candidature réussie – cette approche deviendra rapidement insoluble.

Bien entendu, comme pour de nombreux problèmes en informatique, la solution réside finalement dans une forme de mise en cache. MongoDB et son modèle de données orienté document nous offrent cependant un langage intéressant pour ce type de comptabilité périodique. Étant donné que nous comptons sur une base par utilisateur, nous pouvons utiliser des sous-documents intégrés contenant des noms de propriétés dérivés d'une période de temps. Prenons par exemple un tableau des meilleurs scores prenant en charge les résolutions de la semaine, du mois et du total (sur toute la période).

Pour le nombre de scores de résolution hebdomadaires, nous pouvons nommer les propriétés d'après le numéro de la semaine en cours. Pour lever l'ambiguïté sur plusieurs années, nous pouvons inclure l'année à quatre chiffres dans la clé :

```
# Stocker les scores hebdomadaires dans le sous-
document
        user_doc =
                { "scores_weekly":
                { "2011-01":10,
                "2011-02":3, "2011-06":20
          }
}
```

Pour récupérer le score de cette semaine, nous exécutons simplement la simple recherche dans le dictionnaire suivante :

```
# Récupérer le score de la semaine en cours import
datetime now =
datetime.datetime.utcnow() current_year =
now.year current_week =
now.isocalendar()[1]
# Clés manquantes par défaut pour un score de zéro
user_doc["scores_weekly"].get("%d-%d" %(current_year, current_week), 0)
```

Une telle recherche est incroyablement rapide. Il n'y a aucune agrégation à effectuer. Avec ce modèle, nous pouvons également écrire très rapidement et en toute sécurité. Parce que nous comptons, nous pouvons profiter des modificateurs de mise à jour d'incrémentation et de décrémentation atomique de MongoDB, $inc et $dec. Les opérateurs de mise à jour atomique sont excellents car ils garantissent le sous-jacent

les données sont dans un état cohérent et aident à éviter de mauvaises conditions de concurrence. En particulier lorsqu'il s'agit de facturation, des décomptes précis sont très importants.

Imaginons que nous souhaitions augmenter de 24 le score de l'utilisateur pour cette semaine. Nous pouvons le faire avec la requête suivante :

```
# Mettre à jour le score pour la semaine en cours import
datetime username =
"foouser" now =
datetime.datetime.utcnow() current_year =
now.year current_week =
now.isocalendar()[1]
# Utilisez le modificateur de mise à jour atomique pour incrémenter de
24 dbh.users.update({"username":username}, {"$inc":
    {"scores_weekly.%s-%s" %(current_year, current_week):24}}, sûr = Vrai)
```

Si l'application doit suivre plusieurs périodes, celles-ci peuvent être représentées sous forme de sous-documents supplémentaires :

```
# Stockez les scores quotidiens, hebdomadaires, mensuels et totaux dans le document utilisateur
user_doc =
    { "scores_weekly":
        { "2011-01":10,
        "2011-02":3,
        "2011-06":20
      },
      "scores_daily":
        { "2011-35":2,
        "2011-59":7,
        "2011-83":15
      },
      "scores_monthly":
        { "2011-09":30,
        "2011-10":43,
        "2011-11":24
      },
      "score_total":123
}
```

Bien entendu, dans vos écritures, vous devez incrémenter les décomptes pour chaque période :

```
# Mettre à jour le score pour la semaine en cours import
datetime username =
"foouser" now =
datetime.datetime.utcnow() current_year =
now.year current_month =
new.month current_week =
now.isocalendar()[1] current_day =
now.timetuple( ).tm_yday # Utilisez le modificateur de
mise à jour atomique pour incrémenter de 24
dbh.users.update({"username":username}, {"$inc":

    { "scores_weekly.%s-%s" %(current_year, current_week):24 , "scores_daily.%s-%s" %
        (current_year, current_day):24, "scores_monthly.%s-%s" %(current_year,
        current_month):24,
```

```
            "score_total":24, }

    },
    sûr=Vrai)
```

Dans les cas où vous souhaitez signaler le décompte immédiatement après la mise à jour, cela peut être réalisé en utilisant la commande findAndModify (décrite dans la section précédente) pour renvoyer le nouveau document une fois la mise à jour appliquée.

Ce modèle peut grandement aider avec le comptage à grande vitesse. Si des journaux plus détaillés sont encore nécessaires, par exemple lorsque chaque action a eu lieu, n'hésitez pas à les conserver dans une collection distincte. Ces données récapitulatives sont particulièrement utiles pour les lectures et écritures extrêmement rapides.

CHAPITRE 4

MongoDB avec les frameworks Web

Bien que MongoDB puisse être utilisé dans toutes sortes d'applications, son rôle le plus évident est celui de backend de base de données pour une application Web. De nos jours, un grand nombre de mobiles et de tablettes les applications fonctionnent comme des « gros clients » pour les mêmes API basées sur HTTP que les applications Web basées sur un navigateur ; par conséquent, les applications mobiles et tablettes nécessitent le même type de backend infrastructure de base de données comme des applications Web plus traditionnelles.

De nombreuses organisations et ingénieurs trouvent les avantages de l'architecture orientée document de MongoDB suffisamment convaincants pour migrer des parties, voire des applications entières, d'un SGBDR traditionnel tel que MySQL vers MongoDB. De nombreux célèbres
les entreprises ont construit l'intégralité de leur application à partir de zéro sur MongoDB.

À mon avis, pour la grande majorité des applications Web, mobiles et tablettes, Mon-goDB est un meilleur point de départ que la technologie SGBDR telle que MySQL. Ce chapitre
est une tentative de vous faire décoller en utilisant MongoDB avec trois Python courants
frameworks web : Pylons, Pyramid et Django.

Pylônes 1.x et MongoDB

Pylons est l'un des anciens frameworks Web Python basés sur WSGI, datant de septembre 2005. Pylons a atteint la version 1.0 en 2010 et est considéré comme très stable à l'heure actuelle.
indiquer. En fait, peu de développements sont désormais prévus pour Pylons 1.x ; tous les nouveaux
le développement se déroule dans Pyramid (voir « Pyramid et MongoDB » à la page 49 pour
détails). La philosophie Pylons est exactement à l'opposé de la « taille unique ». Les développeurs d'applications sont libres de choisir parmi les différentes bases de données, modèles et magasins de sessions.
options disponibles. Ce type de cadre est excellent lorsque vous n'êtes pas exactement sûr
de quelles pièces vous aurez besoin lorsque vous commencerez à travailler sur votre application. Si ça tourne
Si vous devez utiliser un système de modèles basé sur XML, vous êtes libre de le faire.

Mis à part l'existence de Pyramid, Pylons 1.x est un framework très performant et stable. Comme
Pylons est si modulaire qu'il est facile d'y ajouter le support MongoDB.

Vous devez d'abord créer un environnement virtuel pour votre projet. Ces instructions supposent que l' outil virtualenv est installé sur votre système. Les instructions d'installation de l' outil virtualenv sont fournies dans le premier chapitre de ce livre.

Pour créer l'environnement virtuel et installer Pylons avec ses dépendances, exécutez les commandes suivantes :

```
virtualenv --no-site-packages myenv cd myenv
source bin/
activer les pylônes
easy_install
```

Nous avons maintenant des pylônes installés dans un environnement virtuel. Créez un autre répertoire nommé comme vous le souhaitez dans lequel créer votre projet Pylons 1.x, remplacez-le par votre répertoire de travail, puis exécutez :

```
coller créer -t pylônes
```

Vous serez invité à entrer un nom pour votre projet, ainsi que le moteur de modèle que vous souhaitez utiliser et si vous souhaitez ou non le mappeur objet-relationnel (ORM) SQLAlchemy. Les valeurs par défaut (« mako » pour le moteur de création de modèles, False pour SQLAl-chemy) conviennent parfaitement à nos besoins, notamment depuis que nous démontrons un logiciel NoSQL.
tabase!

Après avoir exécuté la commande paster create , un répertoire « pylonsfoo » (j'ai choisi « pylonsfoo » comme nom de projet) a été créé avec le contenu suivant :

```
MANIFEST.in
README.txt
development.ini docs

ez_setup.py
pylonsfoo
pylonsfoo.egg-info
setup.cfg
setup.py
test.ini
```

Ensuite, vous devez ajouter le pilote PyMongo en tant que dépendance pour votre projet. Remplacez votre répertoire de travail par le répertoire que vous venez de créer, nommé d'après votre projet. Ouvrez le fichier setup.py présent avec votre éditeur préféré. Modifiez la liste install_requires pour inclure la chaîne pymongo. Votre fichier devrait ressembler à ceci :

```
essayez : depuis setuptools importez la configuration, find_packages
sauf ImportError : depuis
    ez_setup importez use_setuptools use_setuptools()
    depuis setuptools
    importez la configuration, find_packages


setup( nom='pylonsfoo',
    version='0.1',
```

```
    description='',
    author='',
    author_email='', url='',


    install_requires=[ "Pylônes>=1.0", "pymongo",
    ],
    setup_requires=["PasteScript>=1.6.3"],
    packages=find_packages(exclude=['ez_setup']),
    include_package_data=True,
    test_suite='nose.collector',
    package_data={'pylonsfoo' : ['i18n /*/LC_MESSAGES/*.mo']},
    #message_extractors={'pylonsfoo' : [ ('**.py',
    #                'python', Aucun), ('templates/
    #                **.mako', 'mako', {'input_encoding' : 'utf-8'}), ('public/**', 'ignore', Aucun)]},
    #
    zip_safe=False,
    paster_plugins=['PasteScript', 'Pylons'], Entry_points="""
    [paste.app_factory]
     main =
    pylonsfoo.config.middleware:make_app

    [paste.app_install] main
    = pylons.util:PylonsInstaller """,

)
```

Vous devez maintenant récupérer le pilote PyMongo dans votre environnement virtuel. Il est facile de le faire en exécutant : python setup.py develop

Votre application Pylons est maintenant prête à être configurée avec une connexion MongoDB. Tout d'abord, nous allons créer un fichier de configuration pour le développement

```
cp development.ini.sample development.ini
```

Ouvrez ensuite le fichier development.ini dans votre éditeur préféré. Sous la section [app:main] , ajoutez les deux variables suivantes, en modifiant l'URI et les noms de base de données selon ce qui convient à votre configuration :

```
mongodb.url = mongodb://localhost
mongodb.db_name = ma base de données
```

Vous pouvez maintenant essayer de démarrer votre projet avec la commande suivante :

```
coller servir --reload development.ini
```

Vous devriez voir le résultat suivant :

```
Démarrage du sous-processus avec le moniteur
de fichiers Démarrage du serveur dans
le PID 82946. servi sur http://127.0.0.1:5000
```

Si vous ouvrez l'URL http://localhost:5000/ dans un navigateur Web, vous devriez voir la page Pylônes par défaut. Cela signifie que vous avez correctement configuré votre projet. Cependant, nous n'avons pas encore de moyen de parler à MongoDB.

Maintenant que la configuration est en place, nous pouvons indiquer à Pylons comment se connecter à MongoDB et où rendre la connexion PyMongo disponible pour notre application. Pylons offre un endroit pratique pour cela dans <project_name>/lib/app_globals.py. Modifiez ce fichier et modifiez le contenu comme suit : from beaker.cache import

```
CacheManager from beaker.util import
parse_cache_config_options from pymongo import
Connection from pylons import
config


classe Globals (objet):
    """Globals agit comme un conteneur pour les objets disponibles tout au long de la vie de
    l'application
    """


    def __init__(soi, configuration) :
        """Une instance de Globals est créée lors de l'initialisation de l'application et
        est disponible lors des requêtes via la variable 'app_globals'

        """

        mongodb_conn = Connexion(config['mongodb.url']) self.mongodb
        = mongodb_conn[config['mongodb.db_name']] self.cache =
        CacheManager(**parse_cache_config_options(config))
```

Une fois cela configuré, une instance de base de données PyMongo sera disponible pour les actions de votre contrôleur Pylons via l'objet global. Pour démontrer, nous allons créer un nouveau contrôleur nommé « mongodb » avec la commande suivante :

```
contrôleur de pâte mongodb
```

Vous devriez voir un fichier nommé mongodb.py dans le répertoire <project_name>/controllers .
À des fins de démonstration, nous allons le modifier pour incrémenter un document compteur dans MongoDB à chaque fois que l'action du contrôleur est exécutée.

Ouvrez ce fichier avec votre éditeur. Modifiez-le pour qu'il ressemble à ce qui suit (en n'oubliant pas de changer la ligne d'importation from pylonsfoo par le nom que vous avez donné à votre projet) :

```
journalisation des importations

depuis les pylônes, importez app_globals en tant que g, requête, réponse, session, tmpl_context en tant que c, URL
de pylons.controllers.util, abandon de l'importation, redirection

à partir de pylonsfoo.lib.base, importez BaseController, effectuez le rendu

log = journalisation.getLogger(__name__)

classe MongodbController (BaseController) :

    def index(self):
        new_doc = g.mongodb.counters.find_and_modify({"counter_name":"test_counter"}, {"$inc":
            {"counter_value":1}}, new=True, upsert=True , sécurisé =Vrai)
        return "Valeur du compteur MongoDB : %s" % new_doc["counter_value"]
```

Une fois que vous avez enregistré ces modifications, dans un navigateur Web, ouvrez l'URL http://localhost : 5000/mongodb/index. Chaque fois que vous chargez cette page, vous devriez voir un document de la collection counters être mis à jour avec sa propriété counter_value incrémentée de 1.

Pyramide et MongoDB

Pyramid est un framework Web sans opinion issu de la fusion du framework repoz.bfg dans le projet parapluie Pylons (à ne pas confondre avec Pylons 1.x, le framework web). La Pyramide peut être considérée comme un peu un Pylône 2.0 ; il s'agit d'une rupture nette, une toute nouvelle base de code sans compatibilité ascendante au niveau du code avec Pylons 1.x.

Cependant, de nombreux concepts sont très similaires à ceux des anciens Pylons 1.x. Pyramid est l'endroit où se déroulent tous les nouveaux développements, et il offre une couverture et une documentation fantastiques en matière de tests de code. Cette section est uniquement destinée à être une brève introduction à la mise en place d'un projet Pyramid avec une connexion MongoDB. Pour en savoir plus, reportez-vous à l'excellent livre Pyramid et à d'autres ressources disponibles gratuitement en ligne sur http://docs.pylonsproject.org/.

À lui seul, Pyramid n'est qu'un framework, un ensemble de bibliothèques que vous pouvez utiliser. Les projets démarrent plus facilement à partir de ce que l'on appelle un échafaudage. Un échafaudage est comme un squelette de projet qui configure la plomberie et les espaces réservés pour votre code.

Un certain nombre d'échafaudages différents sont inclus avec Pyramid, offrant différentes options de persistance, mappeurs d'URL et implémentations de session. Assez commodément, il existe un échafaudage appelé pyramid_mongodb qui construira pour vous un projet squelette avec le support de MongoDB. pyramid_mongodb vous évite d'avoir à vous soucier de l'écriture du code Glue pour rendre une connexion MongoDB disponible pour le traitement des requêtes dans Pyr-amid.

Comme avec Pylons 1.x, pour commencer à utiliser Pyramid, vous devez d'abord créer un environnement virtuel pour votre projet. Ces instructions supposent que l' outil virtualenv est installé sur votre système. Les instructions d'installation de l' outil virtualenv sont fournies dans le premier chapitre de ce livre.

Pour créer l'environnement virtuel et installer Pyramid et ses dépendances, exécutez les commandes suivantes :

```
virtualenv --no-site-packages myenv cd
myenv
source bin/activer la
pyramide easy_install
```

Prenez note de la ligne source du script bin/activate . Il est important de se rappeler de le faire une fois dans chaque shell pour rendre l'environnement virtuel actif. Sans cette étape, l'installation de votre système Python par défaut sera invoquée, qui n'inclut pas Pyramid.
bloqué.

Pyramid et toutes ses dépendances sont désormais installés dans votre environnement virtuel. Cependant, vous avez toujours besoin de pyramid_mongodb et de ses dépendances comme PyMongo etc. Exécutez la commande suivante pour installer pyramid_mongodb dans votre environnement virtuel :

```
easy_install pyramid_mongodb
```

Avec Pyamid et pyramid_mongodb installés dans votre environnement virtuel, vous êtes prêt à créer un projet Pyramid avec le support de MongoDB. Choisissez un répertoire de projet et un nom de projet. À partir de ce répertoire de projet, exécutez dans le shell :

```
coller créer -t pyramid_mongodb <nom_projet>
```

Après avoir exécuté la commande paster create , un répertoire « mongofoo » (j'ai choisi « mongofoo » comme nom de projet) a été créé avec le contenu suivant :

```
README.txt
development.ini
mongofoo
mongofoo.egg-info
production.ini
setup.cfg
setup.py
```

Les fichiers de configuration par défaut indiquent à Pyramid de se connecter à un serveur MongoDB sur un hôte local et à une base de données appelée « mydb ». Si vous devez changer cela, modifiez simplement les paramètres mon godb.url et mongodb.db_name dans les fichiers INI. Notez que si vous n'avez pas de serveur MongoDB exécuté à l'adresse configurée dans le fichier INI, votre projet Pyramid ne démarrera pas.

Avant de pouvoir exécuter ou tester votre application, vous devez exécuter :

```
python setup.py développer
```

Cela garantira que toutes les dépendances supplémentaires sont installées. Pour exécuter votre projet en mode débogage, exécutez simplement :

```
coller servir --reload development.ini
```

Si tout s'est bien passé, vous devriez voir un résultat semblable à celui-ci :

```
Démarrage du sous-processus avec le moniteur
de fichiers Démarrage du serveur dans
le PID 54019. diffusion sur la vue 0.0.0.0:6543 sur http://127.0.0.1:6543
```

Vous pouvez maintenant ouvrir http://localhost:6543/ dans un navigateur Web et consultez votre projet Pyramid, avec le modèle par défaut. Si vous êtes arrivé jusqu'ici, Pyramid est correctement installé et pyramid_mongodb a pu se connecter avec succès au MongoDB configuré.
serveur.

L' échafaudage pyramid_mongodb configure votre projet Pyramid de telle manière qu'un objet de base de données PyMongo soit attaché à chaque objet de requête . Pour montrer comment l'utiliser, ouvrez le fichier <project_name>/ views.py dans votre éditeur préféré. Il devrait y avoir une fonction Python squelette nommée my_view :

```
def my_view(request):
    return {'project':'mongofoo'}
```

Il s'agit d'une vue pyramidale très simple appelable. Les appelables de la vue pyramidale sont similaires aux actions du contrôleur dans Pylons 1.x et sont l'endroit où se produit une grande partie du traitement des requêtes définies par l'application. Étant donné que les callables de vue reçoivent une instance d'un objet de requête , qui à son tour possède une propriété contenant l' objet de base de données PyMongo , il s'agit d'un endroit idéal pour interagir avec MongoDB.

Imaginez un exemple quelque peu artificiel dans lequel nous souhaitons insérer un document dans une collection appelée « page_hits » à chaque fois que l' appelable vue my_view est exécutée. Nous pourrions faire ce qui suit :

```
import datetime
def my_view(request):
    new_page_hit = {"timestamp":datetime.datetime.utcnow(), "url":request.url}
    request.db.page_hits.insert(new_page_hit, safe=True ) renvoie
    {"project": "mongofoo"}
```

Si vous rechargez maintenant la page Web à l' adresse http://localhost:6543 vous devriez voir une collection appelée « page_hits » dans la base de données MongoDB que vous avez configurée dans votre fichier INI. Dans cette collection, il devrait y avoir un seul document pour chaque fois que la vue a été appelée.

À partir de là, vous devriez être sur la bonne voie pour créer des applications Web avec Pyramid et MongoDB.

Django et MongoDB

Django est probablement le framework Web Python le plus utilisé. Il possède une excellente communauté et de nombreux plugins et modules d'extension. La philosophie Django est à l'opposé des pylônes ou de la pyramide ; il propose un package bien intégré comprenant sa propre base de données et sa propre couche ORM, un système de modèles, un mappeur d'URL, une interface d'administration, etc.

Il existe un certain nombre d'options pour exécuter Django avec MongoDB. Étant donné que Django ORM fait partie intégrante de Django, il existe un projet connu sous le nom de Django MongoDB Engine qui tente de fournir un backend MongoDB pour Django ORM. Cependant, cette approche fait largement abstraction du langage de requête et du modèle de données sous-jacents, ainsi que de nombreux détails de bas niveau abordés au cours du livre. Si vous êtes déjà familier avec Django ORM, que vous aimez l'utiliser et que vous êtes prêt à utiliser un fork de Django, Django MongoDB Engine vaut le détour. Vous pouvez trouver plus d'informations sur le site Web http://django-mongodb.org/.

Notre approche recommandée pour l'instant consiste à utiliser le pilote PyMongo directement avec Django. Sachez cependant qu'avec cette méthode, les composants Django qui dépendent de l'ORM Django (interface d'administration, magasin de sessions etc) ne fonctionneront pas avec MongoDB. Il existe un autre projet appelé Mango qui tente de fournir une prise en charge de session et d'authentification basée sur Mon-goDB pour Django. Vous pouvez trouver Mango sur https://github.com/vpulim/mango.

10gen a mis à disposition un exemple d'application Django avec intégration PyMongo. Cet exemple d'application peut être trouvé sur https://github.com/mdirolf/DjanMon. Nous allons passer en revue l'exécution de l'exemple d'application Django + MongoDB sur votre ordinateur local et examiner comment il configure la connexion MongoDB.

Tout d'abord, téléchargez l'exemple de projet Django. Si les outils de ligne de commande git sont déjà installés, vous pouvez exécuter git clone https://github.com/mdirolf/DjanMon.git. Sinon, cliquez simplement sur le bouton « Télécharger » sur https://github.com/mdirolf/DjanMon.

Afin d'exécuter avec succès l'exemple d'application, vous devrez créer un environnement virtuel Python avec Django, pymongo et PIL installés. Comme pour Pylons et Pyramid, vous devrez d'abord installer l' outil virtualenv sur votre système. Les détails sur la façon de procéder sont traités dans le premier chapitre de ce livre. Une fois virtualenv installé, choisissez un répertoire dans lequel stocker virtual env, puis exécutez ce qui suit
commandes shell dedans :

```
virtualenv --no-site-packages djangoenv cd djangoenv
source bin/activer
pip installer django pymongo
PIL
```

Cela créera votre environnement virtuel, l'activera puis y installera Django, le pilote Py-Mongo et la bibliothèque de manipulation d'images PIL (requise par l'application de démonstration). En supposant que tout cela ait réussi, vous êtes prêt à démarrer l'exemple de serveur de développement d'application. Notez que l'exemple d'application s'attend à ce qu'un serveur MongoDB s'exécute sur localhost.

Nous pouvons maintenant exécuter l'application de démonstration Django de 10gen. Remplacez votre répertoire de travail actuel par votre copie du projet « DjanMon ». Il devrait y avoir un fichier appelé man age.py dans le répertoire de travail actuel. L'application peut être exécutée avec le serveur de développement Django avec la commande :

```
python manage.py serveur d'exécution
```

Vous devriez voir une sortie sur la console comme celle-ci :

```
Validation des modèles...

0 erreur trouvée
Django version 1.3, en utilisant les paramètres 'DjanMon.settings'
Le serveur de développement s'exécute sur http://127.0.0.1:8000/ Quittez le
serveur avec CONTROL-C.
```

Vous pouvez maintenant ouvrir un navigateur Web et visiter http://localhost:8000/ et consultez l'application de démonstration ! L'application vous permet de créer des messages simples (éventuellement avec des images jointes) qui sont conservés dans MongoDB.

Examinons le fonctionnement de l'exemple d'application. Jetez un œil au fichier status/view.py. C'est ici que la connexion MongoDB est créée et que la majeure partie de la logique de l'application est stockée. Dans leur exemple d'intégration Django + MongoDB, 10gen adopte une approche différente des autres décrites dans ce chapitre. Ils créent une base de données PyMongo dans

la portée globale du module de vues, plutôt que de l'attacher à des requêtes d'objets comme dans Pyramid ou d'en faire un framework global comme dans Pylons 1.x :

```
importer datetime
importer une
chaîne importer
aléatoire importer des
types MIME importer cStringIO en tant que StringIO

depuis PIL import Image
depuis django.http import HttpResponse depuis
django.http import HttpResponseRedirect depuis django.shortcuts
import render_to_response depuis pymongo.connection import
Connexion depuis pymongo import DESCENDING import
gridfs


db = Connexion().sms
```

Cette approche est simple et fonctionne bien pour une démo. Cependant, dans les projets Django plus importants avec plusieurs applications installées (dans cet exemple, il y a une seule application installée – elle est nommée « statut ») cela nécessiterait qu'un pool de connexions PyMongo distinct soit maintenu pour chaque application. Cela entraîne un gaspillage de connexions MongoDB et du code dupliqué.
Au lieu de cela, il serait recommandé de créer la connexion en un seul endroit et de l'importer dans tous les autres modules nécessitant un accès.

Cela devrait suffire à vous permettre de commencer à créer votre application Django MongoDB.

Aller plus loin

Dans ce livre, nous avons essayé de vous donner une solide compréhension de la façon d'exploiter MongoDB dans des applications du monde réel. Vous devez avoir une bonne compréhension de la façon de modéliser vos données, d'écrire des requêtes efficaces et d'éviter les problèmes de concurrence tels que les conditions de concurrence et les blocages. Il existe un certain nombre d'autres sujets avancés pour lesquels nous n'avions pas de place dans ce livre, mais qui méritent néanmoins d'être étudiés lors de la création de votre application. Notamment, map-reduce permet de calculer efficacement les agrégats.
Le partage vous permet de faire évoluer votre application au-delà de la mémoire disponible d'une seule machine. GridFS vous permet de stocker des données binaires dans MongoDB. Les collections plafonnées sont un type spécial de collection, qui ressemble à un tampon circulaire et est idéale pour les données de journal.
Avec ces fonctionnalités à votre disposition, Python et MongoDB sont des outils extrêmement puissants à avoir dans votre boîte à outils lors du développement d'une application.

A propos de l'auteur

Niall O'Higgins est un consultant en logiciels spécialisé dans l'informatique mobile, tablette et cloud. Ses réalisations incluent la conception et la mise en œuvre du backend de la plateforme Catch.com à l'aide de MongoDB, Python et Pylons. Catch est l'un des plus populaires applications sur Android. Avant Catch, il était ingénieur logiciel chez Metaweb Technologies, où il a travaillé sur Freebase.com (maintenant propriété de Google). Il est le fondateur et organisateur du San Francisco Python Web Technology Meet-up, PyWebSF, et le Bay Area Tablet Computing Group, nous avons des tablettes. Il a publié pas mal un peu de logiciel Open Source - contribuant notamment à OpenBSD - et prend fréquemment la parole lors de conférences et d'événements. Vous pouvez le trouver sur Twitter sous le nom de @niallohig-gins.