NobleProg

Table of Contents

MongoDB概要

- リレーショナルデータベースとMongoDBの違い
- MongoDBは高速なドキュメント指向データベース
- 「行」より柔軟なモデルである「ドキュメント」を使用する
    - ドキュメントは具体的にはBSON (JSONライクなフォーマット)を表している。
- NoSQL の一種
    - SQLを使用しない。
    - ただし、MongoDBは他のNoSQLよりも分析クエリが柔軟に行える。SQLに近い分析、たとえば集計などが手軽にできる。
- スキーマレス
    - RDBMSと比較して柔軟
- トランザクションがない(または不完全)

    - お金を扱うようなシステムには向かない
    - ただし、MongoDB 4.0からは複数ドキュメントにまたがるトランザクションが使える
- 様々なインデックスをサポート
- 水平スケールアウトがしやすい
- MongoDB(ドキュメント指向)以外のNoSQLには何があるか
    - Key Value Store (KVS)
        - データをKey と Valueの形式で持つ
        - 基本はKeyでの完全一致検索
        - Redis, Memcached
    - 列指向データベース
        - 列単位でデータを取り出すときに効率的
        - 大量なデータを高速に集計するのに優れる
        - Cassandra, HBase
    - グラフ型データベース
        - グラフ構造を備えたデータベース。グラフはノード、エッジ、プロパティの３要素によって、ノード間の関係性を表現
        - Neo4j

[演習]
次のうち、MongoDBに向いているシステムはどれか。

＊ 入出金管理
＊ 映像データ、音声データの保存
＊ ログ収集・分析基盤

Getting Started

RDB	MongoDB
データベース	データベース

テーブル	
行	
列	

- ドキュメントはデータの基本単位。RDBMSでのrow(行)に相当。
- コレクション = テーブル
- すべてのドキュメントには _id が含まれる。
    - コレクション内では一意となるID
- JavaScriptシェル (mongo)で管理およびデータ操作を行う

$ mongo
$ mongo localhost:27017

Database

- データベースはコレクションのグループを保持する
- 1つのデータベース＝1つのアプリケーション
- 複数のデータベースを単一インスタンスで利用可能
- 異なるユーザーやアプリケーションに対応するためにデータベースは分離される
- データベース名は英数字の文字列で、大文字小文字を区別し、最大64バイト、空の文字列は許可されない
- データベース名はファイルシステム上のファイルとなる
- 特別なデータベース:
    - admin: ユーザー情報などを格納する。管理者のみがアクセス可能。
    - local: これに属するコレクションはレプリケートされない。インスタンス固有のデータベース
    - config: シャーディング時に利用

- namespace(名前空間)はデータベースとコレクション名(完全修飾されたコレクション名)を連結したもの
    - 最大120バイト (version4.4以降は255バイト)
    - 例) hogeDatabase.barCollection

データベースの作成・使用はuse
> use NewDB
> db.myNewCollection1.insertOne({ x: 1 })
データベース一覧を表示して 作成の確認
> show dbs

[演習] 下記コマンドはエラーとなる理由はなぜか。

> use Hello World

[演習] "hr"データベースの"employee"コレクションのネームスペースは何になるか

Collection

- RDBMSでのテーブル。ドキュメントのグループを保持する。
- 動的スキーマ
    - 型を指定しなくても、動的にデータ型が決定される。
    - {"company" : "NobleProg"}
        - 文字列型
    - {"age" : 5}
        - 数値型
- コレクション名は文字列
    - ただし、$から始まるのは不可。

- 同じ種類の情報を一つのcollectioで保持。
    - クエリで取得する関連情報をコレクションに含めてもよい
- コレクションの作成

    - コレクションはドキュメントを登録した時点で作成される

> db.myNewCollection2.insertOne({ x: 1 })

- コレクションの確認
    - show collectionsで選択中のデータベースに属するコレクションを表示

> show collections

Document

- ドキュメントは、関連する値を持つ順序付きのキーの集合。
- BSON
    - MongDBはドキュメントをBSONというJSONに近い形式で保持する
    - バイナリ型JSON
    - JSONにくらべストレージ容量及びスキャン速度に効率的な設計
- key は文字列であり、UTF-8を使用できる
    - $から始まるのは不可。
- type-sensitive {"age" : 3}, {"age" : "3"}.
- case-sensitive {"age" : 3}, {"Age" : 3}
    - 大文字・小文字を区別します。
- ドキュメントには、重複したキーを含めることはできない。
    - 例： {"company" : "NobleProg", "training" : "MongoDB", "company" : "NobleProg", }
- キーと値のペアの順序は {"x" : 1, "y" : 1} != {"y" : 1, "x" : 1} です。
    - 順番は通常問題ではないが、MongoDB はキーの順番を入れ替えることができる。
- (プログラム言語で扱う際、)ドキュメントの表し方はプログラミング言語によって異なる
    - map; hash・・・Perl, Ruby； dictionary・・・Python

ドキュメントの例：

```
{
"_id" : ObjectId("545a414c7907b2a255b156c5"),
"Name" : "Sean Connery",
"Nationality" : "Great Britain",
"BirthDate" : ISODate("1930-08-25T00:00:00Z"),
"BirthYear" : 1930,
"Occupation" : [
"Actor",
"Director",
"Producer"
],
"Movie" : [
{
"_id" : ObjectId("545a5f167907b2a255b156c7"),
"Title" : "Dr. No"
},
{
"_id" : ObjectId("545a5f317907b2a255b156c8"),
"Title" : "From Russia with Love"
},
{
"_id" : ObjectId("545a5ed67907b2a255b156c6"),
"Title" : "Never Say Never Again"
}
],
"BirthPlace" : {
"Country" : "United Kingdom, Scotland",
"City" : "Edinburgh"
}
}
```


MongoDB のインストール 手順

- Installation version 3.6 on RedHat / CentOS
    - https://docs.mongodb.com/v3.6/tutorial/install-mongodb-on-red-hat/
- Installation on Windows
    - http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/
- Installation on Ubuntu
    - http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

ここでは、CentOS に MongoDB3.6をインストールする手順を掲載する。

```
#  yumリポジトリ追加
   (rpm packageをダウンロードする方法もある )

$ cd /etc/yum.repos.d/
$ vi MongoDB.repo

[mongodb-org-3.6]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/3.6/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.6.asc

# インストール
$ sudo yum install -y mongodb-org

# version 確認
$ mongo --version

# systemctlでmongod起動 (System V init で行う方法もあります)
$ sudo systemctl start mongod

# 状態確認
$ sudo systemctl status mongod
```

# システム再起動後もサービス起動させる場合
$ sudo systemctl enable mongod

# mongo シェル確認
$ mongo --host 127.0.0.1:27017

上記でインストールされたMongoDBのフォルダ構成

/etc/mongod.conf - 設定ファイル

/var/log/mongodb/mongod.log - ログファイル

/var/lib/mongo - データの格納先ディレクトリ

/var/run/mongodb/mongod.pid - プロセスIDを保持しているファイル

/usr/lib/systemd/system/mongod.service - サービス定義ファイルです。

[演習]

* プロセスIDを保持しているファイルの中身を表示してください。
* LINUXのプロセス一覧からMongoDBのプロセスを確認してください。
そのプロセスIDがファイルの値と同じことを確認してください。

CRUD

Create

> use NobleProg
データの登録

> db.products.insert({ item: "card", qty: 15 })

データの登録(変数にセットしてから登録)
> person = {"Name" : "Barack Obama", "Nationality" : "United States", "Age": 59}
> db.people.insert(person)

データの複数登録
> db.products.insert(
[
{ _id: 11, item: "pencil", qty: 50, type: "no.2" },
{ item: "pen", qty: 20 },
{ item: "eraser", qty: 25 }
]
)

Read

全件取得
> db.people.find()

一件のみ取得
> db.products.findOne()

条件を指定して検索
> db.products.findOne(
{ item: "pen"}
)

項目を指定して検索
0・・非表示、1・・表示
> db.products.find({}, {"_id": 0, "qty": 1})

$gt より大きい
$gte 以上
$lt より小さい

$lte 以下

qty > 30
> db.products.find({qty: {$gt: 30}});
結果をフォーマットして表示 db.people.find().pretty()

Update

Barak Obamaの職業を設定。(先程の変数personを使用してアップデート)
> person.Occupation = "Politician"
> db.people.update({"Name" : "Barack Obama"}, person)

"eraser" の 個数を２７に変更
db.products.update({name:"eraser"},{$set: {qty: 27}});

Upsert
Upsertは、なければ登録、あれば更新してくれる便利な機能

該当するドキュメントが無いので、登録となる。
db.people.update(
{Name: "Kei Nishikori"},
{"Name": "Kei Nishikori", "Nationality": "Japan", "Height": "168"},
{upsert: true}
)

該当するドキュメントがあるので、更新となる。
db.people.update(
{Name: "Kei Nishikori"},
{"Name": "Kei Nishikori", "Nationality": "Japan", "Height": "178"},
{upsert: true}
)

確認
db.people.find({Name: "Kei Nishikori"})

Delete
条件を指定してデータ削除
db.people.deleteOne({"Name" : "Barack Obama"})
db.people.deleteMany({"Name" : "Barack Obama"})

全件削除
db.people.deleteMany({})
確認
db.people.findOne()

[演習]

- testデータベースのcomposerコレクションに、下記のドキュメントを登録してください。

{"Name": "Beethoven", "BirthYear": 1770},

{"Name": "Bach", "BirthYear": 1685},

{"Name": "Mozart", "BirthYear": 1756},

{"Name": "Debussy", "BirthYear": 1862}

- 次のようにドキュメントを更新してください。

NameがBeethovenのドキュメントに"From": "Germany" のフィールドを追加

- 次の条件にあてはまるドキュメントを検索してください。

BirthYear < 1800

- 次の条件にあてはまるドキュメントを削除してください。

BirthYear > 1800

- composerコレクションのすべてのドキュメントをfind()で表示してください。ただし、BirthYear、_idフィールドは表示しないでください

Data Type

- MongoDBは様々なデータ型をサポートします。
  - null {"x" : null}
  - boolean {"x" : true}
  - number (by default 64-bit floating point numbers) {"x" : 1.4142}
    - 4-byte integers {"x" : NumberInt(141)}
    - 8-byte integers {"x" : LongInt(141)}
    -
  - string (any UTF-8 character) {"x" : "NobleProg"}
  - date (エポックからのミリ秒を保持) {"x" : new Date()}
  - 正規表現 {"x" : /bob/i}
  - 配列 {"x" : [1.4142, true, "training"]}
  - オブジェクト埋め込みドキュメント {"x" : {"y" : 100}}
  - ObjectId {"x" : ObjectId("54597591bb107f6ef5989771")}
    - _idで使用される
  - バイナリデータ (for non-UTF-8 strings)
  - code {"x" : function() {/*..*/}}
    - JavaScriptの関数を格納。

[アンチパターン] 何重ものネスト

[演習] さまざまなデータ型のドキュメントを登録してみましょう。

_id / ObjectId

- MongoDB のすべてのドキュメントは _id を持つ
- 任意の型を指定できるが、デフォルトはObjectId
- コレクション内でユニークな値
- ObjectIdは軽量で簡単に生成できるように設計されている
- 生成規則
- 12バイト(24桁の16進数からなる)
  - 4バイトのUnixエポックからの経過秒数(Unix時間)
  - 3バイトのマシンID
  - 2バイトのプロセスID
  - 3バイトのインクリメンタルカウンタ(開始番号はランダム)
- _id はドキュメントに存在しない場合、自動的に生成される

MongoDB Shell

シェルは"mongo"で起動する
$ mongo
MongoDB shell version: X.X.X
connecting to: test

ホスト名、ポート番号、データベース名を指定してシェルを起動
$ mongo HostName:PortNumber/DatabaseName
$ mongo localhost:27017/test

どのデータベースにも接続しない場合、--nodbをつけて起動
$ mongo --nodb

次の例では、デフォルトポートの localhost 上で実行している MongoDB インスタンスへの新しい接続をインスタンス化し、
getDB() メソッドを使ってグローバル変数「db」にNobleProg を設定している
> conn = new Mongo("localhost:27017")
connection to localhost27017

```
> db = conn.getDB("NobleProg")
NobleProg
```

ヘルプの使用

- mongo - JavaScriptシェル内で、ヘルプはJavaScriptのオンラインドキュメントとして提供されている
- MongoDBに特化した機能のために組み込みのヘルプを使用することができます
- 関数名を括弧なしで入力すると、関数の実態がわかる

```
> help
db.help() help on db methods
db.mycoll.help() help on collections methods
...
exit quit mongo shell
```

関数名を括弧なしで入力すると、関数が何をしているかがわかる

```
> db.NobleProg.stats
function ( scale ){
return this._db.runCommand( { collstats : this._shortName , scale : scale } );
}
>
> db.NobleProg.stats()
{ "ok" : 0, "errmsg" : "Collection [test.NobleProg] not found." }
>
```

> db.stats // 関数をカッコなしで実行すると、関数の中身をみることができる

function (scale) {

```
return this.runCommand({dbstats: 1, scale: scale});
}
> db.stats() // 関数の実行
{
"db" : "NobleProg",
"collections" : 2,
"views" : 0,
"objects" : 1,
"avgObjSize" : 63,
"dataSize" : 63,
"storageSize" : 49152,
"numExtents" : 0,
"indexes" : 2,
"indexSize" : 36864,
"ok" : 1
}
```

[演習]

* mongoシェルで、helpを表示しましょう。

*** helpを参考に、 どのようなログがあるか表示してみましょう。**

*** globalのログを表示させてみましょう。**

スクリプトの実行

- mongoシェルでJavaScriptファイルを実行することが可能
  - mongo script.js
- スクリプト内ですべてのグローバル変数 (例えば "db" など) にアクセスできる
- シェルヘルパー (例: "show collections") はファイルからは動作しません; 有効な JavaScript で同等な処理を使用してください (例: "db.getCollectionNames()")
- スクリプト実行時に "MongoDB shell version..." など、接続時の文言を非表示にするには --quiet を使用する
- Mongo Shell から直接スクリプトを実行するにはload() を使用する
- .mongorc.js
  - .mongorc.js ファイルをユーザのホームディレクトリに配置しておくと mongo 起動時に自動的に .mongorc.js を読み込み実行する。シェル実行中に共通で使いたい変数や関数をあらかじめ定義しておくことができる
  - .mongorc.js を読み込みたくない場合、 --norc オプションを付与して mongo を起動する
  - プロンプトのカスタマイズにも使用される
- .dbshell
  - mongo シェルで実行されたコマンドの履歴が保存される
  - ファイルはホームディレクトリ直下に自動生成される

スクリプトの実施
$ mongo script.js
MongoDB shell version: 4.0.6
connecting to: test
script.js was executed successfully!
$

--queitで接続時の文言を非表示

```
$ mongo --quiet script.js
script.js was executed successfully!
$
```

シェルに接続してからスクリプトを実行

```
$ mongo
MongoDB shell version: 4.0.6
connecting to: test
> load("script.js")
script.js was executed successfully!
```

Shell HelperとScriptでの記述の比較

Shell Helpers	JavaScript Equivalents
show dbs, show databases	db.adminCommand('listDatabases')
use <db>	db = db.getSiblingDB('<db>')
show collections	db.getCollectionNames()
show users	db.getUsers()
show roles	db.getRoles({showBuiltinRoles: true})

show log <logname>	db.adminCommand({ 'getLog' : '<logname>' })
show logs	db.adminCommand({ 'getLog' : '*' })

it	cursor = db.collection.find()
	if (cursor.hasNext()){
	cursor.next();
	}

[実演]

* NobleProgデータベースのコレクションをすべて表示するスクリプトを作成し、実行してください。

* mongo シェルで実行されたコマンドの履歴を表示してください。
* cursorを使用してNobleProgデータベースのpeopleコレクションのドキュメントをすべて表示するスクリプトを作成し、実行してください。
ヒント：
　cusrorにセットされたドキュメントの表示：printjson(cursor.next())

外部エディタによる変数の編集 / 閲覧

- 変数を外部エディタで編集することができる
- 表示したい変数が長い場合などにも、外部エディタで表示すると重宝する

- EDITOR="/usr/bin/gedit"
- EDITOR="/usr/bin/vi"
- EDITOR="c:\\windows\\notepad.exe"

```
$ mongo
> EDITOR="/usr/bin/gedit"
> use NobleProg
switched to db NobleProg
> person = db.people.findOne()
{ "_id" : ObjectId("54568445cfc7c83518fa5430"), "Name" : "Sean Connery" }
> edit person
> person
{ "_id" : ObjectId("54568445cfc7c83518fa5430"), "Name" : "Sean Connery", "Nationality" : "Great Britain" }
> db.people.save(person)
```

[演習]

*サーバーのステータスをmongoで表示してみましょう。サーバーのステータスは

db.serverStatus()で表示できます。

*上記の結果が長いので少々見にくいようです。サーバーのステータスの結果を外部エディタで表示してみましょう。

サーバーの設定

設定ファイル

- 設定ファイルでMongoDBの設定を行う。
  - YAMLフォーマットの設定ファイル
- パッケージマネージャでインストールした場合、/etc/mongodb.confに配置されている
- 次のようにファイルを指定する

mongod --config <path>
mongod -f <path>

* 設定の有効化にはmongodの再起動が必要。

設定ファイルの例

net:
port: 27017
bindIp: 127.0.0.1
operationProfiling:
mode: slowOp
slowOpThresholdMs: 10
storage:
dbPath: c:\data\db
wiredTiger:
engineConfig:
cacheSizeGB: 1
systemLog:
destination: file
path: c:\data\logs\mongodb.log
security:
authorization: enabled
keyFile: c:\data\config\keyfile.txt
replication:
replSetName: training
oplogSizeMB: 128

ストレージエンジン

- ストレージエンジンは、データがディスクにどのように保存されるかを管理する

    - データベースとハードウェアの間のインターフェース
- MMAPv1ストレージエンジン
    - 最初のストレージエンジン
    - コレクションレベル ロック
    - 書き込み時、ロックを取得
- WiredTigerストレージエンジン
    - バージョン3.0の新機能
    - 3.2 以降のデフォルト
    - ほとんどのユースケースでパフォーマンスが向上
    - ドキュメントレベルロック
    - 圧縮（データおよびインデックス）
    - 書き込み時、ロックしない
- インメモリストレージエンジン
    - メモリ内にデータを保持することで高速なクエリパフォーマンスを実現
    - そのインスタンスでは永続化されず、mongodプロセスが再起動すると書き込まれた全てのデータは消える

MMAPv1

- mongod --storageEngine mmapv1
    - version 4.2以降は廃止

WiredTiger

- --storageEngine wiredTiger
- データは B-Trees に保存される
    - 最初はドキュメントが未使用領域に書き込まれる
    - その後バックグラウンドで残りのデータとマージされる
    - B-tree
        - ツリー状の構造で、1つのノードから3つ以上の子ノードを

持っているのが特徴のデータ構造。検索性能が優れており、RDBのインデックスによく使用される。

- WTは2つのキャッシュを使用する
  - WT インターナルキャッシュ
    - 50% (RAM - 1 GB)の50% または256MBの大きい方
  - オペレーティングシステムキャッシュ
- データの永続性を確保するために、チェックポイントと組み合わせて、ライトアヘッド(書き込み優先)のトランザクションログを使用する
- MongoDB はチェックポイントをディスクにコミットします
  - 前回のチェックポイント終了から60秒後、または
  - WTキャッシュにダーティデータ(変更されたデータ)が多い(2ギガバイトを超えた)場合
- ドキュメントレベルロック
  - WT にはロックはないが、優れた同時実行プロトコルがある
  - 書き込みはスレッド数に応じてスケールします。
- 圧縮(コレクションごとに個別に設定可能)
  - snappy (デフォルト。高速、バランスのとれたストレージ効率と処理要件)
    - block_compressor=snappy
  - zlib (より多くの CPU を使用してsnappyより高い圧縮率)
    - block_compressor=zlib
  - 圧縮しない
    - block_compressor=none

storageEngineをWiredTigerにし、圧縮をzlibに指定してemailコレクションを生成
db.createCollection("email", { storageEngine: { wiredTiger: { configString: 'block_compressor=zlib' }}})

認証と認可

- MongoDBはロールベースのアクセス制御を採用

    - ユーザーには1つまたは複数のロールが付与され、それによってデータベースリソースと操作へのユーザーのアクセス権が決定される
- ロール
    - リソース上で指定されたアクションを実行するための権限を付与
    - 各権限はロール内で明示的に指定されるか
    - もしくは別のロールから継承される
- Privileges(権限)
    - 指定されたリソースとそのリソースで許可されたアクションで構成される
    - リソース： データベース、コレクション、コレクションのセット、またはクラスタ
    - アクション： リソースで許可されている操作
    - 権限の継承
        - ロールは他のロールを保持できる。そのロールは含まれているロールのすべての権限を継承する。
- MongoDB は以下の認証をサポート
    - SCRAM(チャレンジレスポンス認証)
        - MongoDBのデフォルトの認証方式。チャレンジレスポンス： パスワードによって認証するが、パスワードを直接やり取りするわけではない。
    - x.509 証明書
        - 公開鍵認証、TLS/SSL

(設定例)
net:
ssl:
mode: requireSSL
PEMKeyFile: /etc/ssl/mongodb.pem
CAFile: /etc/ssl/caToValidateClientCertificates.pem

- LDAP proxy (enterprise edition)
- Kerberos認証 (enterprise edition)
- メンバーの内部認証
    - (Securityの欄で説明)
- 認証はデフォルトでは無効
    - mongod --auth により有効化
        - 設定ファイルの場合
        - security:
          authorization: enabled

    - 少なくとも一つのスーパーユーザーアカウントを作成する。
- ユーザはデータベースに属しており、属するデータベースで認証されなければならない。
    - adminデータベースで作成されたユーザーは、すべてのデータベースに対して操作を行うことができる
- adminデータベースで作成されたロールはadminデータベース、他のデータベース、クラスタリソースに適用される権限を含めることができ、adminデータベースだけでなく他のデータベースのロールからも継承することができる
- adminデータベース以外で作られたロールはそのデータベースに適用される権限のみを含めることができ、そのデータベース内の他のロールからのみ継承することができる
- adminデータベースで作成されたユーザーは、すべてのデータベースに対して操作を行うことができます。
- built-inロール: read、readWrite、dbAdmin、userAdmin、dbOwner 等

ユーザー管理の実習

実際にユーザーを作成し、ユーザー管理の様々な操作を実行してみましょう。

- スーパーユーザー作成
    - adminデータベースに属し、rootロールを付与
- テストユーザー作成
    - Roleの付与
- カスタムロールの作成

スーパーユーザー作成

現在のデータベースのユーザー一覧表示
> show users

> use admin
> db.createUser({user : "admin", pwd : "NobleProg", roles : ["root"]})

adminユーザーでログインしてみましょう。

$ mongo -u admin -p --authenticationDatabase admin
> use admin
> db.auth("admin", "NobleProg") # MongoDBに接続してから認証を実行する方法

テストユーザー作成

次に、testデータベースに属するユーザーを追加してみましょう。
> mongo -u admin -p --authenticationDatabase admin
> use test
> db.createUser({user: "testuser", pwd: "NobleProg", roles: ["dbOwner"]})
Successfully added user: { "user" : "testuser", "roles" : ["dbOwner"] }

ユーザーが追加されていることを確認します。
> db.getSiblingDB("admin").system.users.find().pretty()

testuserでログインできることも確認してみましょう。
$ mongo test -u testuser -p --authenticationDatabase test

テストユーザーの情報を表示
> db.getUser("testuser")

dbOwnerロールの情報を表示
> db.getRole("dbOwner")

dbOwnerロールの情報と有する権限を表示
> db.getRole("dbOwner", {showPrivileges: true}) {

testuserにロールを付与
> db.grantRolesToUser("testuser", [{role: "read", db: "dept"}])
> db.grantRolesToUser("testuser", [{role: "readWrite", db: "dept"}])
testuserの情報をもう一度みてみましょう

> db.getUser("testuser")

ロールの剥奪
> db.revokeRolesFromUser("testuser", [{role: "read", db: "dept"}])
testuserの確認
> db.getUser("testuser")

dbAdminロールを付与
> db.grantRolesToUser("testuser", [{role: "dbAdmin", db: "dept"}])
ユーザー一覧を表示 (show usersと同等)
> db.getUsers()

ユーザー削除
> db.dropUser("testuser")
true
削除を確認してみましょう。
> db.getUser("testuser")
null
> show users
> db.getUsers()
[ ]
> db.logout()

カスタムロールの作成

> use admin
> db.auth("admin", "NobleProg")
> db.createUser({user: "monitoringuser", pwd: "NobleProg", roles: []})
statsWatcherというロールを作成する
> db.createRole({role: "statsWatcher", privileges: [{resource: {"anyResource": true}, actions: ["serverStatus"]}], roles: []})
> db.getRole("statsWatcher", {showPrivileges: true})

ロールをユーザーに付与 > db.grantRolesToUser("monitoringuser", [{role: "statsWatcher", db: "admin"}])

monitoringuserでログインし、serverstatusを実行できるかを試してみましょう。

> db.serverStatus()

MongoDBの監視

- mongotop
  - コレクション毎に、操作に利用した時間が表示される
  - topコマンドのようなコマンドであり、処理に時間がかかっているコレクションを処理時間順に見ることができる
  - 高負荷時にどれだけ多くread,writeされているがわかるので、上から順に見直しをするとよい
  - フィールド
    - ns: database.collection
    - total: 総計時間
    - read: 読み込み時間
    - write: 書き込み時間
    - timestamp: データのタイムスタンプ
  -
  - 実行例 以下のような出力が1秒ごとに出力される。
    - $ mongotop --host localhost --port 27102
      ns total read write 2020-11-26T23:33:21+09:00
      db1.collection1 10ms 1ms 9ms
      db1.collection2 8ms 0ms 8ms
      db2.collectionA 0ms 3ms 0ms
      ns total read write 2020-11-26T23:33:22+09:00
      sample-db.collection01 9ms 2ms 7ms
      sample-db.collection02 8ms 0ms 8ms
      adb2.collectionA 0ms 4ms 0ms
  -
-
- mongostat
  - mongodもしくはmongosインスタンスに対して統計情報を定期的に表示する。

- LINUXのvmstatのようなコマンド。

- 実行例
  - mongostat --host localhost --port 27102 --discover
  - --discoverオプションをつけるとレプリカセット・シャードクラスタの
    全てのインスタンスの状態を表示
  - host insert query update delete getmore command dirty used flushes vsize res qrw arw net_in net_out conn set repl time
    localhost:27101 *0 *0 *0 *0 0 5|0 0.0% 0.0% 0 4.89G 26.0M 0|0 1|0 537b 61.1k 6 np_rep SEC Nov 26 23:31:10.467
    localhost:27102 no data received

    localhost:27101 *0 *0 *0 *0 0 3|0 0.0% 0.0% 0 4.89G 26.0M 0|0 1|0 422b 60.1k 6 np_rep SEC Nov 26 23:31:11.462
    localhost:27102 *0 *0 *0 *0 0 4|0 0.0% 0.0% 0 4.89G 26.0M 0|0 1|0 943b 60.6k 6 np_rep SEC Nov 26 23:31:11.460
- **insert**
  1 秒間にデータベースに挿入されるオブジェクトの数。アスタリスク (*など) が続く場合、データムはレプリケートされた操作を参照しています。
  query
  1 秒あたりのクエリ操作数。
  update
  1秒あたりの更新操作数。
  delete
  1秒あたりの削除操作数。
  getmore
  1 秒あたりの get more (すなわちカーソルバッチ) 操作の数。
  command
  1秒あたりのコマンド数です。セカンダリシステムでは、mongostatはパイプ文字(|など)で区切られた2つの値を、ローカル|複製されたコマンドの形で表示します。
  flushes
  WiredTiger Storage Engine の場合、flushes は各ポーリング間隔の間にトリガーされる WiredTiger チェックポイントの数を指します。
  dirty
  WiredTiger Storage Engine のみ。wiredTiger.cache.tracked dirty bytes in cache / wiredTiger.cache.maximum bytes configuredで計算される、ダーティバイトを持つWiredTigerキャッシュのパーセンテージ。
  used
  WiredTiger Storage Engine のみ。使用中の WiredTiger キャッシュの割合を、現在のキャッシュ内の wiredTiger.cache.bytes / wiredTiger.cache.maximum bytes で計算します。

vsize
最後のmongostat呼び出し時にプロセスが使用したメガバイト単位の仮想メモリの量です。
res
最後のmongostat呼び出し時にプロセスが使用したメガバイト単位の常駐メモリの量です。
locked
グローバルな書き込みロックにかかっている時間の割合。
qr
MongoDB インスタンスからのデータ読み込みを待つクライアントのキューの長さ。
qw
MongoDB インスタンスからのデータ書き込みを待つクライアントのキューの長さ。
ar
読み取り操作を実行しているアクティブなクライアントの数。
aw
書き込み操作を実行しているアクティブなクライアントの数。
netIn
MongoDB インスタンスが受信したネットワークトラフィックの量 (バイト単位)。
これには mongostat 自体からのトラフィックも含まれます。
netOut
MongoDB インスタンスが送信したネットワークトラフィックの量 (バイト単位)。
これには mongostat 自体からのトラフィックも含まれます。
conn
開いている接続の総数。
セット
該当する場合はレプリカセットの名前。
レプリカ
メンバのレプリケーション状態。
PRI: primary
SEC: secondary
REC: recovering
UNK: unkown
PTR: mongos process("router")
ARB: arbiter

- MongoDB Monitoring Service (MMS)
  - 監視やバックアップなど、自動運用管理をしてくれるサービス
  - オンプレ/クラウド版

- MMSエージェントを監視用サーバーにインストールし、MMSと接続する必要がある
- 他にMuninやNew Relicのプラグインも用意されており、MuninやNew Relicで監視することもできる

サーバの統計情報を詳細に表示

　接続数，挿入や更新，クエリ，削除の合計数、メモリ使用量、ネットワークの情報などを表示

```
> db.serverStatus()
> db.runCommand("serverStatus")
```

次のように一部の情報のみ表示することもできる
現在のメモリ使用量を表示
db.serverStatus().mem
db.runCommand({ serverStatus: 1}).mem
ネットワーク
db.serverStatus().network
各種メトリクス
db.serverStatus().metrics

レプリカセットの統計情報を表示
レプリカセットの他のメンバーが送信したハートビートパケットから得られたデータを使用して、レプリカセットの現在の状態を出力

```
> rs.status()
> db.runCommand("replSetGetStatus")
```

実行中のコマンドを表示

```
> db.currentOp()
```

データベースの統計情報を表示

```
> db.stats()
```

コレクションの統計情報を表示

```
> db.collection.stats()
```

プロファイラ

- Profilerはクエリパフォーマンスの調査に使用される
- データベース プロファイラは、実行中の mongod インスタンスに対して実行されたデータベース コマンドの詳細情報を収集する。これには、設定コマンドや管理コマンドだけでなく、CRUD操作も含まれる。
- プロファイラは収集したすべてのデータを system.profile コレクションに書き込む
- プロファイリングレベル
    - 0 - プロファイリングなし
    - 1 - 「遅い」操作のみを含む
    - 2 - すべての操作を含む

現在のプロファイリングの状態を表示

> db.getProfilingStatus()
was: 現在のプロファイリングレベル
"slowms": 遅いクエリを表示するしきい値
"sampleRate": サンプルレート(0 ~ 1.0)
sampleRate フィールドは、プロファイル化する必要がある低速操作の割合を示す。

100ms以上かかっているクエリのみ出力
> db.setProfilingLevel(1, 100)

出力先のdb.system.profileから、スロークエリを表示してみましょう
millisの降順でソートすることで、遅いクエリの順に表示
> db.system.profile.find().sort({millis : -1}).pretty()

インデックス

- データベースのインデックスは、本のインデックスのように動作する
- MongoDB のインデックスはリレーショナルデータベースとほぼ同じ動作をする
- インデックスのため、読み取りは速くなるが、書き込みは遅くなる
- 1つのコレクションにおけるインデックスの最大個数は64個
- インデックスが無い場合、全件検索となる

- 検索に時間がかかる
- サーバー負荷も増大

インデックスの管理

- インデックス作成
- db.collection.createIndex(key1:1,key2:1,...})
  - 指定する値により昇順・降順がきまる
  - 1: フィールドの値の昇順にインデックスを作成
    -1: フィールドの値の降順にインデックスを作成

> db.categories.createIndex({CategoryID: 1})

- インデックスの削除
  - db.collection.dropIndex("インデックス名"), .dropIndexes()

> db.categories.dropIndex()

- インデックスの確認
  - db.collection.getIndexes()

> db.categories.getIndexes()

- オプション: background, name
  - background をtrueにすると、バックグラウンドでインデックスを作成する
  - インデックス名は自動で生成されるが、nameを指定すると、インデックス名を指定する

nameでインデックス名を指定し、バックグラウンドでインデックスを作成

> db.categories.createIndex(

{CategoryID: 1},

{name: "Index_CategoryID", background: true}

)

確認

> db.categories.getIndexes()

[演習] usageコレクションのusageIDにインデックスを作成してください。名前はindex_usageIDとし、バックグラウンド実行してください。

インデックスなしのコレクション

インデックスの無いコレクションの作成
for (var i=1; i<=1000000; i++) {
db.visitors.insert({
"i" : i,
"visitor" : "visitor_"+i,
"score" : Math.floor(Math.random()*10+1),
"date" : new Date()
})
}
確認
> db.visitors.count()

一つのフィールドのインデックス

インデックスを使用しないクエリ実行を行います。下記を実行してみましょう。
db.visitors.explain().find({"visitor" : "visitor_330"})

db.visitors.explain().find({"visitor" : "visitor_99999"}).limit(1)

visitor項目にインデックスを作成
db.visitors.createIndex({"visitor" : 1})

db.visitors.explain().find({"visitor" : "visitor_99999"})

Compoundインデックス

- Compound Index(複合インデックス)
  - 単一のインデックス構造で、コレクションのドキュメント内の複数のフィールド [1] への参照を保持
- 作成方法
  - db.collection.createIndex({ <field1>: <type>, <field2>: <type2>, ... })

下記は、エラーとなる(大量のデータに対してソートにはインデックスが必要)
> db.visitors.find().sort({"score" : 1, "visitor" : 1})

compondインデックスを作成
db.visitors.createIndex({"score" : 1, "visitor" : 1})

再度、クエリを実行
> db.visitors.find().sort({"score" : 1, "visitor" : 1})
{ "_id" : ObjectId("5fa75198be64f00a72f69318"), "i" : 100, "visitor" : "visitor_100", "score" : 1, "date" : ISODate("2020-11-08T02:02:00Z") }
{ "_id" : ObjectId("5fa75198be64f00a72f6969c"), "i" : 1000, "visitor" : "visitor_1000", "score" : 1, "date" : ISODate("2020-11-08T02:02:00.238Z") }
{ "_id" : ObjectId("5fa7519abe64f00a72f6b9c4"), "i" : 10000, "visitor" : "visitor_10000", "score" : 1, "date" : ISODate("2020-11-08T02:02:02.428Z") }
{ "_id" : ObjectId("5fa751b0be64f00a72f81963"), "i" : 100015, "visitor" : "visitor_100015", "score" : 1, "date" : ISODate("2020-11-08T02:02:24.465Z") }

explainによりインデックススキャンとなる確認
> db.visitors.find().sort({"score" : 1, "visitor" : 1}).explain()
"stage" : "IXSCAN",

下記も、インデックスScanとなる
> db.visitors.explain(true).find({"score" : 10}).sort({"visitor" : -1})

db.visitors.explain(true).find({"score" : {"$gte" : 10, "$lte" : 20}})
db.visitors.explain(true).find({"score" : {"$gte" : 10, "$lte" : 20}}).sort({"visitor" : 1})

* {key1 : 1, key2 : 1, key3 : 1}のインデックスがある場合、{key1 : 1}は必要ない

  →{"score" : 1, "visitor" : 1}のインデックスがあるため、scoreへのインデックスが効い

> db.visitors.explain(true).find().sort({"score":1})

Compounインデックス (2)

- インデックスは、フィールドへの参照を昇順 (1) あるいは降順 (-1) のソート順で保持する。単一フィールドインデックスの場合は、キーのソート順は関係ないが、複合インデックスの場合は、ソート順が問題になることがある。
- {"score" : -1, "visitor" : 1}。
  - -1は降順、1は昇順
- {"score" : -1, "visitor" : 1} = {"score" : 1, "visitor" : -1}。
  - .sort({ score: -1, visitor: 1 }) ←インデックスが効く
  - .sort({ score: 1, visitor: -1 }) ←インデックスが効く
  - .sort({ score: 1, visitor: 1 }) ←インデックスが効かない

- カバードインデックス (indexOnly : true または totalDocsExamined = 0 (totalKeysExamined > 0 の場合)
  - カバードクエリとは、次の場合にデータベースにアクセスせずにインデックスから値を取得し、結果を高速にかえすこと
    - クエリ内のすべてのフィールドがインデックスの一部
    - クエリで返されるフィールドはすべて同じインデックスの中にある
- {key1 : 1, key2 : 1, key3 : 1}のインデックスがあれば、{key1 : 1}と{key1 : 1, key2 : 1}を作成する必要はない。

カバードインデックスの例

下記はtotalDocsExaminedが0となる。これはカバードクエリーとなり、非常に高速となる

この時、表示フィールドに関して、_idは0(非表示)にする必要がある。_idはindexにふくまれていないため。

> db.visitors.explain(true).find({score : 8, visitor: "visitor_100146"},{score: 1, visitor: 1, _id: 0})

配列やサブドキュメントのインデックス

- マルチキーインデックス
  - 配列内の要素に対するインデックス
- 通常のインデックスと同様に動作
- インデックスごとに1つの配列フィールドのみ
  - ２つ以上の配列フィールドには使用不可
  - たとえば、次のドキュメントに対して{ a: 1, b: 1 }のインデックスは作成できない
  - { _id: 1, a: [1, 2], b: [1, 2], category: "AB - both arrays" }

配列に対してインデックスを作成する例
 データ準備
> db.survey.insert({item: "ABC", ratings: [2, 5, 9] })

インデックスの作成方法は通常と同様。配列であれば、自動的にマルチキーインデックスとなる。
> db.survey.createIndex({ ratings: 1 })

ratings配列に対する検索はインデックスが効いている
> db.survey.find({ratings: 5}).explain()

オブジェクトを要素として持つ配列のインデックス
入れ子オブジェクトを持つ配列フィールドのインデックスを作成することができる

入れ子オブジェクトを持つ配列フィールドのインデックスの例
　データ準備
> db.inventory.insert([{

```
item: "abc",
stock: [
{ size: "S", color: "red", quantity: 25 },
{ size: "S", color: "blue", quantity: 10 },
{ size: "M", color: "blue", quantity: 50 }
]
},
{
item: "def",
stock: [
{ size: "S", color: "blue", quantity: 20 },
{ size: "M", color: "blue", quantity: 5 },
{ size: "M", color: "black", quantity: 10 },
{ size: "L", color: "red", quantity: 2 }
]
},
{
item: "ijk",
stock: [
{ size: "M", color: "blue", quantity: 15 },
{ size: "L", color: "blue", quantity: 100 },
{ size: "L", color: "red", quantity: 25 }
]
}])
```

stock.size、stock.quantityに対してインデックスを作成

```
> db.inventory.createIndex( { "stock.size": 1, "stock.quantity": 1 } )
```

下記のクエリはインデックスを利用

```
> db.inventory.find( { "stock.size": "S" } ).explain()
> db.inventory.find( { "stock.size": "S", "stock.quantity": { $gt: 20 } } ).explain()
```

[演習] inventoryコレクションのcolorに対してインデックスを作成してください。

インデックス詳細

- カーディナリティが高く、選択率が低い項目ほどインデックスが有効
- カーディナリティ
    - DBのフィールドに含まれるデータの種類の高低の度合いを意味する。カーディナリティが低い/データの種類が少ないものの例：性別。
- 選択率
    - 条件となる値によってドキュメントがどの程度まで絞り込めるかを示す数値
    - キー値のように、一意に絞り込めるものであれば選択率は非常に低くなる
    - 不等号演算子 $nin や $ne は、インデックスの大部分にマッチすることが多いので、インデックスの効果があまりでない場合もある

- explain()
    - クエリの実行計画を表示する
    - インデックスが使用されているか
    - COLLSCAN: フルスキャン。インデックスが使用されていない
    - IXSCAN: インデックスが使用されたスキャン
    - winningPlan: オプティマイザに採用された実行計画
    - rejectPlans: 採用されなかった実行計画
- .hint()でindexの使用を強制する

ageフィールドのインデックスを使用するように強制
> db.users.find().hint({ age: 1 })
インデックス名を指定してインデックスの使用を強制
> db.users.find().hint("age_1")

Partialインデックス

- Partialインデックス(部分インデックス)は、一定の条件を満たすフィールドにインデックスを作成する
- クエリで絞り込む条件が明確な場合、部分インデックスによりインデックスのサイズを削減できる
- partialFilterExpressionオプションでフィールドと条件を指定する
    - 次の条件を指定できる
        - 等式 (フィールド: 値、または $eq 演算子の使用)。

- exists: true
- $gt, $gte, $lt, $lte
- $type
- $and

データの準備
> db.users.insert({ userID:1613471, name:"Yamada", age: 31 });
> db.users.insert({ userID:1236523, name:"Sato", age: 25 });
> db.users.insert({ userID:8348345, name:"Suzuki", age: 18 });
> db.users.insert({ userID:5844352, name:"Hashimoto", age: 57 });
> db.users.insert({ userID:9922560, name:"Okada", age: 49 });
> db.users.insert({ userID:4562842, name:"Watanabe", age: 43 });
> db.users.insert({ userID:4499033, name:"Takahashi", age: 43 });

部分インデックスを作成する。age>40に対してインデックスを作成。
> db.users.createIndex({ userID: 1 }, { partialFilterExpression: { age: { $gt: 40 } } })

まず、部分インデックスが効いていない例
　部分インデックスで設定した条件と異なるので、フルスキャンとなる
> db.users.find({ userID:{ $gt: 400000 }}).explain()
"winningPlan" : {
"stage" : "COLLSCAN",

age>40 の条件にすると、IXSCANとなり、インデックスが効く
> db.users.find({ userID:{ $gt: 400000 }, age: { $gt: 40 } }).explain()

"winningPlan" : {
"stage" : "FETCH",
"filter" : {
"age" : {
"$gt" : 40
}
},
"inputStage" : {
"stage" : "IXSCAN",

age>50もインデックスが効く
> db.users.find({ userID:{ $gt: 400000 }, age: { $gt: 50 } }).explain()
"inputStage" : {
"stage" : "IXSCAN",

age>30とすると部分インデックスで設定した範囲外なので、インデックスが効かない
> db.users.find({ userID:{ $gt: 400000 }, age: { $gt: 30 } }).explain()
"winningPlan" : {
"stage" : "COLLSCAN",

Nullになりえる項目のユニークキー

- ユニーク制約を付けたいが、nullの可能性もある場合
- version3.2より前のバージョンではsparse indexで対応。version3.2以降は部分インデックスで対応するのがよい。

xという項目にuniqueインデックスを付与した場合
> db.pi.insert([{y:1, x:1}, {y:1, x:2}, {y:1, x:3}, {y:1}])
> db.pi.createIndex({x:1}, {unique:1})
下記は、xの項目が無いのでエラーとなってしまう。
> db.pi.insert({y:1})

一旦インデクッスを削除
> db.pi.dropIndexes()

xにpartialFilterExpressionで存在する場合のみuniqueインデックスを付与

> db.pi.createIndex(
{ x: 1 },
{ unique: true, partialFilterExpression: { x: { $exists: true } } }
)
> db.pi.createIndex({x:1}, {unique:1, sparse:1})
下記は、xが存在しないがエラーとならない。
> db.pi.insert({y:1})

下記は、重複キーエラーとなる
> db.pi.insert({y:5, x:1}) WriteResult({
"nInserted" : 0,
"writeError" : {
"code" : 11000,
"errmsg" : "E11000 duplicate key error collection: test.pi index: x_1 dup key: { : 1.0 }"
}
})

sparse_indexのxを指定しているので、xのフィールドを持たないドキュメントはhitしない
> db.pi.find({"x" : {"$ne" : 2}}).hint({"x" : 1})

下記は、xのフィールドを持たないドキュメントもHitする
> db.pi.find({"x" : {"$ne" : 2}}).hint({"$natural" : 1})

Capped コレクション

- データの上限を設定したコレクション
  - 一定期間のみデータが必要なケースに向いてる
  - ユースケース(例)： ログ用のコレクション
  - 書き込み速度が20%ほど早い
  - 定期的に手動でデータを削除するようなオペレーションを削減できる
- 最初の挿入が行われる前に作成されなければならない
- 一定のサイズまたはドキュメント数(データ数)を上限とする
- ドキュメントの禁断の操作: 削除、更新(サイズが大きくなる場合)
- Cappedコレクションの制限
  - シャーディング不可

    - 上限を途中で変更できない
    - データの削除ができない(自動削除のみ)
- ソート: $natural : 1 (または -1)
    - データベースに保持されたままの順序をあらわす

Cappedコレクションは下記のように作成する
>db.createCollection("コレクション名",{capped:true,size:<データサイズの上限>, max: <ドキュメント数の上限>})

capped_collectionという名で、上限データサイズ 100000Byteのコレクションを作成
> db.createCollection("capped_collection", {"capped" : true, "size" : 100000})

既存の固定コレクションをCappedコレクションに変換するには、convertToCappedを使用
> db.people.copyTo("capped_collection2")
> db.runCommand({"convertToCapped" : "capped_collection2", "size" : 100000})

Tailableカーソル

- 結果が出尽くした時点で終了しないカーソル。
    - tail -f コマンドをイメージしてください
    - (通常のカーソルは結果が出尽くしたらクローズする)
- Cappedコレクションにのみ使用可能
- コレクションに新しいドキュメントが登録されると、Tailable Cursorは結果を取得する
- indexは使用不可。
    - 最初の結果表示は時間がかかるが、新しく追加されるデータは高速
- ユースケース：リアルタイム処理

参考：https://yut.hatenablog.com/category/MongoDB

TTL インデックス (time-to-live)

- TTLインデックスでは、ドキュメントごとにタイムアウトを設定することができる。
    - expireAfterSeconds で指定した秒数より古くなったときに削除される
- 単一フィールド(Date型、またはDate型を含む配列フィールド)にのみ作成可能です。

データの準備
> db.ttl_collection.insert({"User" : "user1", "LastUpdated" : new Date()})

TTL Indexeを作成。オプション指定は {"expireAfterSeconds" : <秒数>}
> db.ttl_collection.createIndex({"LastUpdated" : 1}, {"expireAfterSeconds" : 30})
> db.ttl_collection.find()

Full-Textインデックス

- 組み込みの多言語サポートによるクイックテキスト検索
- テキストインデックスは大きなサイズになりがち
    - ドキュメント書き込みのスループットに影響を与える
- db.collection.createIndex({"key" : "text"})
- weight
    - weight(重み)で重要度を調整することができる
    - MongoDB は重みをかけてマッチ数を算出し、結果を合計する。この合計を使って MongoDB がドキュメントのスコアを計算する。
    - デフォルトの重みは1

> use sample
customersコレクションに対してtextインデックスを生成
> db.customers.createIndex({"ContactTitle": "text"})

次の例は、weightsでCompanyNameに重みをつけている
> db.customers.createIndex({"CompanyName" : "text", "ContactTitle" : "text"}, {"weights" : {"CompanyName": 2}})

文字列の項目すべてにテキストインデックスを作成したい場合は、$**を使用することができます(ワイルドカードtextインデックス)
> db.customers.createIndex({"$**" : "text"})

$textを使用して、テキストインデックスを生成したフィールドに対してテキスト検索が可能。

スペースで区切られたキーワードのいずれかにHitするドキュメントを検索(OR検索)
> db.customers.find({$text : {$search : "sales manager"}})

ダブルクォートでキーワードを囲むことで完全一致で検索できます
> db.customers.find({$text : {$search : "\"Sales Manager\""}})

「-」でキーワードを除外します。次の例は、Salesというキーワードをもつドキュメントを除外します.
> db.customers.find({$text : {$search : "-Sales Manager"}})

スコアを表示
> db.customers.find({$text : {$search : "Sales Manager"}}, {score : {$meta : "textScore"}})

スコアを表示し、スコア順にソート
> db.customers.find({$text : {$search : "Sales Manager"}},
{score : {$meta : "textScore"}}).sort({ score: { $meta: "textScore" } })

Geospatial 地理空間インデックス

2d インデックス

- 二次元平面上の点として格納されたデータに対して使用するインデックス

データ準備
> db.dots.insert([{Name:"A", location:[10, 5]}, {Name:"B", location:[17, -5]}, {Name:"C", location:[0, 2]}, {Name:"D", location:[-3, -3]}])

2dインデックス作成
db.dots.createIndex({location:"2d", type:1})

0,0座標に近い点を検索
db.dots.find({location:{$near:[0,0]}})

2dスフィア インデックス

- 地球のような球体(sphere)の幾何学を計算するクエリをサポート。
- GeoJSONオブジェクトとして格納されたデータに対するインデックス。
    - GeoJSON・・・以下２つのプロパティを持つ。
        - type: Point, LineStringなど。参照：https://docs.mongodb.com/manual/reference/geojson/#
        - coordinates: 座標
- 参考： 地理空間クエリ： http://docs.mongodb.org/master/reference/operator/query-geospatial/
- 距離を計算する: http://docs.mongodb.org/master/tutorial/calculate-distances-using-spherical-geometry-with-2d-geospatial-indexes/

都市をあらわす地点を登録
db.cities.insert({Name:"Rzeszów", "location": {"type":"Point", "coordinates":[22.008606,50.040264]}})
db.cities.insert({Name:"Warszawa", "location": {"type":"Point", "coordinates":[21.0123237,52.2328474]}})
db.cities.insert({Name:"Wrocław", "location": {"type":"Point", "coordinates":[17.0342894,51.1174725]}})
db.cities.insert({Name:"Kraków", "location": {"type":"Point", "coordinates":[19.9012826,50.0719423]}})
db.cities.insert({Name:"Kielce", "location": {"type":"Point", "coordinates":[20.6156414,50.85404]}})
地理空間インデックスの作成
db.cities.createIndex({location:"2dsphere"})

"Kielce"に近い地点を検索 0~200000メートルの距離の都市を表示
db.cities.find({location:{$near:
{$geometry: {type:"Point", coordinates:[20.6156414, 50.85404]}, $minDistance: 0, $maxDistance:200000}
}})

-> 近い順に結果が表示される。

ある地点に近い地点を検索。近い順に結果を表示。距離をdist表示。
db.runCommand({geoNear: "cities", near: [20.6156414, 50.85404], spherical: true, distanceMultiplier: 6378.1})

指定した多角形(ポリゴン)の範囲内の地点を検索
db.cities.find({location: {$geoWithin:
{$geometry: {type: "Polygon", coordinates: [[[22.008606,50.040264], [21.0123237,52.2328474], [19.9012826,50.0719423], [22.008606,50.040264]
]]}}
}})

[演習] あるECサイトでユーザーは商品コレクションを検索する際に、評価フィールド(0~10)が0.9以上のもの、と指定するユーザーが大半であった。
インデックスのサイズを削減しつつ、検索を高速化するにはどういったインデックスを作成するのが適しているか。
[演習] 社内の情報を一時的に登録する"一時掲示板"コレクションのデータは、書き込み終了後、24時間で削除したい。
どのようなインデックスを作成すべきか。

セキュリティ

プロダクション環境の運用では、セキュリティを向上するために、最低限以下の設定が必要となる。

メンバー間の内部認証

- レプリカセットやシャードクラスタのメンバー間の認証(内部認証)はKeyFileか x.509 証明書のどちらかを使う。
- KeyFIle
    - キーファイル は Base64セットの文字列（[a-zA-Z0-9/+] の64種類）を使った 6〜1024文字 のファイル
    - --keyFile オプションでKeyFileへのパスを指定。もしくは設定ファイルで指定。

KeyFile生成例
$ openssl rand -base64 756 > <path-to-keyfile>
$ chmod 400 <path-to-keyfile>

(設定ファイルでの指定)
security:
  keyFile: <path-to-keyfile>

ポート番号の変更

- 「MongoDBのデフォルトポート番号を狙ったアクセスが頻発している」という注意喚起が警視庁から行われたことがあり、デフォルトポート番号の変更が推奨される。
    - --port オプションでポート番号を指定

(設定ファイルでの指定)
net:
  port: <port-number>

IPの設定

- bindIpを設定すると、MongoDBへのアクセス元を制限することができる。
  - --bind_ip <IP Address>
  - 複数指定の場合は、カンマ区切り
    - [127.0.0.1, 192.168.10.100]
  - 0.0.0.0 ですべてのIPアドレスからアクセス可能に。（開発用途など）

(設定ファイルでの指定)
net:
  bindIp: <IP Address>

認証設定を有効化する

- MongoDBはデフォルトでは認証機能がOffになっているが、プロダクション環境ではOnにすることが推奨される。

(設定ファイルでの指定)
security:
  authorization: enabled

レプリケーション

- レプリケーションとは、データを他のサーバーに同期して、冗長化し、可用性を高めること。
- セカンダリから読み取ることで、読み取りの負荷分散

- データセンターが異なれば災害対策にもなる
- すべての本番環境に推奨
- レプリカセットは、1つのプライマリサーバと多数のセカンダリのセット
- MongoDBのレプリケーションは非同期
- プライマリサーバはクライアントのリクエストを処理
- 自動フェイルオーバー
    - プライマリーがクラッシュした場合、セカンダリーから新しいプライマリーが選択され、システムが稼働し続ける

検証用の簡易レプリカセット

- ReplSetTest でレプリカセットの機能の確認ができる。
- ポート 20000, 20001, 20002を使用
    - データベースはデフォルトのディレクトリ (/data/db) に保存される

ReplSetTestによりテスト用レプリカセットを構築し、プライマリにデータ投入してセカンダリにデータがコピーされるかを検証

$mkdir /data/

$mkdir /data/db

```
$ mongo --nodb
> replicaSet = new ReplSetTest({"nodes" : 3})
> replicaSet.startSet()
> replicaSet.initiate()
```

起動しているインスタンスを確認
> ps aux | grep mongo

プライマリとなっているかを確認

```
$ mongo --nodb
> conn0 = new Mongo("localhost:20000")


> db0 = conn0.getDB("test")
> db0.isMaster()
```

データを投入してみる
```
> for (i=0; i<100; i++) { db0.repl.insert({"x" : i}) }
> db0.repl.count()
```

Slaveに接続
```
> conn1 = new Mongo("localhost:20001")
> db1 = conn1.getDB("test")
```

マスターかどうかを確認
```
> db1.isMaster()
```

下記はエラーとなる。Secondary の古いデータの読み込みを防ぐためにデフォルトでSecondaryへのリクエストは拒否される
```
> db1.repl.count()
(前後略)"errmsg" : "not master and slaveOk=false",
```

Secondaryへのリクエストを許可
```
> db1.setSecondaryOk()
```
　もしくはrs.secondaryOk()

データがレプリケートされている確認
```
> db1.repl.count()
```

Slaveに書き込みはできるか？
db1.repl.insert({"x" : 1})

masterをシャットダウン
> db0.adminCommand({"shutdown" : 1})
> db1.isMaster()

シャットダウン (20001/20002)
$ mongo localhost:20001
> use admin
> db.shutdownServer()

より実戦に近いセットアップ例

- この例では、3つのメンバーでレプリカを起動する方法を示す。
- 同じサーバー上で構成する。hostを指定することで、複数のマシンに簡単に変更することができる。
- レプリカセット名は任意の utf-8 文字列 (この例では np_rep) 。
- mongod起動時に レプリカセットの名前を指定する。
    - この例では --replSet np_rep。
- レプリカセットの各メンバーは、すべてのメンバーと接続できなければならない。

事前にフォルダ作成

mkdir /data/np_rep

mkdir /data/np_rep

mkdir /data/np_rep/rep0

mkdir /data/np_rep/rep1

mkdir /data/np_rep/rep2

3つのメンバーを起動する ※forkはバックエンドでの起動

```
$ mongod --port 27100 --replSet np_rep --dbpath /data/np_rep/rep0 --logpath /data/np_rep/rep0.log --fork --bind_ip 127.0.0.1 --logappend
$ mongod --port 27101 --replSet np_rep --dbpath /data/np_rep/rep1 --logpath /data/np_rep/rep1.log --fork --bind_ip 127.0.0.1 --logappend
$ mongod --port 27102 --replSet np_rep --dbpath /data/np_rep/rep2 --logpath /data/np_rep/rep2.log --fork --bind_ip 127.0.0.1 --logappend
```

- レプリカセットを設定するには Mongo Shell を使用
- 設定ドキュメントを準備する
    - _id キーはレプリカセットの名前
    - members はレプリカセットに含まれるすべてのサーバのリスト
- サーバに設定をデータと共に送信
- このサーバは他のメンバーの設定を変更する
- メンバーはプライマリを専任し、クライアントのリクエストの処理を開始する

MongoDBにアクセス

```
> mongo localhost:27100
```

設定確認。初期化前なので、下記はエラーとなる

```
rs.conf()
```

レプリカセットの初期化

```
> np_rep_config = {
... "_id" : "np_rep",
... "members" : [
... {"_id" : 0, "host" : "localhost:27100"},
```

... {"_id" : 1, "host" : "localhost:27101"},
... {"_id" : 2, "host" : "localhost:27102"}
...]
... }
>
> rs.initiate(np_rep_config)

rs.initiateは下記のようにしてもよい(設定ファイルを使用しない例)

>rs.initiate(

{
... "_id" : "np_rep",
... "members" : [
... {"_id" : 0, "host" : "localhost:27100"},
... {"_id" : 1, "host" : "localhost:27101"},
... {"_id" : 2, "host" : "localhost:27102"}
...]
... }
)

設定確認。_id: レプリカセット名 membersにレプリカセットのメンバーの情報を表示。

rs.conf()

primaryに昇格した場合、シェルに下記のように表示される。昇格前はnp_rep:SECONDARY>
np_rep:PRIMARY>

マスターになったかを確認

rs.isMaster()

priamryにデータ書き込みをしてみる

> use test

> db.rp1.insert({x:1})

セカンダリの情報をみてみる

$ mongo localhost:27102

もしくは> db = (new Mongo("localhost:27102")).getDB("test")
> db.isMaster()

データがレプリケートされている確認

> db.setSecondaryOk()

> db.rp1.find()

レプリカセットの終了方法

※ --evalはshellを開かずにコマンド実行する。

$ mongo --port 27100 --eval 'db.adminCommand("shutdown")'
$ mongo --port 27101 --eval 'db.adminCommand("shutdown")'

$ mongo --port 27102 --eval 'db.adminCommand("shutdown")'

もしくはそれぞれのメンバーに対して下記を実施

> use admin

> db.shutdownServer();

もしくは データベースを指定して終了

$ mongod --shutdown --dbpath /data/np_rep/rep0

$ mongod --shutdown --dbpath /data/np_rep/rep1
$ mongod --shutdown --dbpath /data/np_rep/rep2

レプリカセットのプライマリへのシェル接続

- レプリカセットのプライマリにシェルで接続
    - --host replicaSetName/hostを指定すると、プライマリに接続する
    - mongo --host replicaSetName/host1[:porthost1],host2[:porthost1],host3[:porthost3],etc

いずれかのメンバーを指定すると、プライマリに接続する
> mongo --host np_rep/localhost:27102
np_rep:PRIMARY>

rsヘルパー

- rs はレプリケーションヘルパー関数を含むグローバル変数
- rs.help() - すべての関数のリストを表示
- rs.initiate(np_rep_config)
    - レプリカセットを初期化する
    - adminCommand のラッパー。db.adminCommand({"replSetInitiate" : np_rep_config}) と同等

> rs.help()
メンバーの追加
> rs.add("localhost:27103")

アービターの追加
rs.addArb("localhost:27104")

メンバーの削除
> rs.remove("localhost:27103")
> rs.remove("localhost:27104")
設定の確認
> rs.config()
設定を変数にセット > np_rep_config = rs.config()
設定をエディターで編集 (事前に EDITOR変数をセット。 EDITOR="/usr/bin/vim") > edit np_rep_config
既存のレプリカセットの設定を上書きする(Primaryに対して行うこと。Secondaryに行う場合は、force: trueが必要） > rs.reconfig(np_rep_config)

[演習]

一つのセカンダリのpriorityを0に設定してください。設定にはrs.reconfigを使用します。rs.config()で確認後、設定を元にもどしてください。

レプリカセットの構成

- レプリカセットの構成
  - レプリカセットには最大50台までのメンバーを含めることができ、そのうち最大7台までが投票権を持つことができる。 投票権を持つサーバー台数は奇数になるよう設計する。 足りない場合はアービターを追加して調整する。
- フォールトトレランスの検討
  レプリカセットの耐障害性(フォールトトレランス)とは、レプリカセットが利用できなくなっても、プライマリを選出するのに十分なメンバーをセットに残しておくことができるメンバーの数のこと。言い換えれば、セット内のメンバーの数と、プライマリを選出するために必要な投票メンバーの過半数との差。プライマリがなければ、レプリカセットは書き込み操作を受け付けることができない。

メンバーの数	新しいプライマリを選出するのに必要なメンバーの数	Fault Tolerance
3	2	1
5	3	2
7	4	3
9	5	4

選挙(Election)

- レプリカセットは、どのセットメンバーがプライマリになるかを決定するためにElection(選挙)を実施する。
- Electionは次の場合に行われる
    - 新しいノードが追加されたとき
    - セカンダリがプライマリに１０秒間接続できないとき
        - レプリカセットのメンバーは、2秒ごとにお互いにハートビート（ping）を送信する。10秒以内にハートビートが返ってこないメンバーは、アクセスできないメンバーとしてマークされる。
    - rs.stepDown() や rs.reconfig() などのメンテナンスメソッドを実施したとき
- サーバーが投票を行う際、より高い優先度（priority）が設定されたサーバーが優先的にプライマリに選任される。 優先度が低いサーバーが選任されている状態で、より高い優先度のサーバーが有効になった場合、再び優先度が高いサーバーがプライマリとなるよう投票が行われる。
- 投票権を有する会員
  members[n].votesの設定が1に設定されているレプリカ・セット・メンバーはすべて、選挙で投票する。選挙で投票しないようにするには、そのメンバーの members[n].votes 設定の値を 0 に変更する。(投票権を持たないメンバーのpriorityは0でなければならない)

  以下の状態の投票権を持つメンバーのみが投票権を持つことができる。

PRIMARY
SECONDARY
STARTUP2
RECOVERING
ARBITER
ROLLBACK

- Election時はデータ書き込みはできない。読み込みは、セカンダリから読み込むように設定している場合は可能。
- ネットワークパーティション(分断)
    - ネットワークパーティションはプライマリを少数のノードを持つパーティションに分離する可能性がある。プライマリがレプリカセットの中のマイノリティー(少数)のノードしか見えないことを検出すると、プライマリはプライマリとしてステップダウンしてセカンダリになる。独立して、(自分自身を含む)過半数のノードと通信できるパーティション内のメンバが、新しいプライマリになるための選挙を行う。

メンバーの設定

- **arbiter**
-
    - ArbiterはDBのデータは保持せず、単にPrimary死亡時の昇格投票のみを行う。実データの読み書きが一切なく、負荷は非常に低い。
    - Arbiterは必須ではなく、通常PrimaryとSecondaryのメンバー数の合計が偶数になってしまう場合に数合わせとして参加させるメンバー
    - Arbiterのmongod プロセスは --replSet オプションと空のデータディレクトリで起動する
    - Arbiter追加コマンド
        - > rs.addArb("server:port")
        - > rs.add({"_id" : 7, "host" : "server:port", "arbiterOnly" : true})
    - アービターは通常、最大一つのみ
    - アービターは一度追加されると通常のmongodに変更することはできない。逆に通常のmongodをアービターに変更することもできない
    - 可能な限り、アービターの代わりに通常のデータメンバーを使用したほうがよい

```
> mongod --port 27104 --replSet np_rep --logpath /data/np_rep/rep4.log --fork --bind_ip 127.0.0.1 --logappend
> rs.addArb("localhost:27104")
```

- **priority**
    - 優先度が高いほどプライマリになる確率が高い
    - 優先度の範囲は0から100まで（デフォルトは1)
    - 優先度0のメンバーはプライマリになることができない(パッシブ・メンバー)
- **hidden**
    - hiddenメンバーはprimaryのコピーを保持するが、プライマリになることはない
    - クライアントアプリケーションからは非表示となる
    - 選挙の際、投票はできる
    - hiddenメンバーはクライアントのリクエストを処理しない
    - バックアップやレポート作成に便利
    - priority : 0 、 hidden : true を指定することでhiddenメンバーとなる
    - db.isMaster()の結果からはhiddenメンバーは非表示になる
        - rs.status(), rs.config()では表示される
    - シャード化したクラスターではmongosはhiddenメンバーと通信しない

[演習]
一つのセカンダリをhiddenメンバーにし、rs.isMaster()でそのメンバーが表示されないことを確認してください。

[演習]
priority 0 でhiddenのメンバーに対して、下記の操作のうち不適切なのはどれか。
* テキスト検索
* 分析クエリー
* データー括登録

- **Delayedメンバー**
    - 遅延セカンダリは指定された秒数だけプライマリからのデータコピーが遅れる
    - ユーザの操作ミス等に対する保険的な利用が可能
        - ヒューマンエラーなどが起こった際は、Delayedメンバーをレプリカセットから切り離すことでデータの保全ができる

    - 設定方法
        - slaveDelay : <秒数>
        - priority : 0
            - プライマリになることはできない
        - hidden: true
            - アプリケーションからは見えない/クエリできない
    - 想定する復旧時間より大きい秒数をセットすること
    - oplogのキャパシティより小さい秒数をセットすること

一日(86400秒)分ディレイしたメンバーを追加
PRIMARY> member = {
"host" : "127.0.0.1:27106",
"priority" : 0,
"slaveDelay" : 86400,
"hidden" : true
}
PRIMARY> rs.add(member)

- **インデックスを作成しないメンバー**
    - buildIndexes: falseに設定すると、インデックスを作成しないメンバーとなる
    - バックアップサーバに便利
        - クライアントからのクエリを処理しないメンバー
    - 設定方法
        - buildIndexes : false
        - priority : 0
    - 途中で変更することはできない

rs.add({
"host" : "127.0.0.1:27107",

"priority" : 0,
"buildIndexes" : false,
"hidden": true
})

Sync (同期)

MongoDB では、データの同期には 2 つの形式(Initial Sync、レプリケーション)がある。
初期同期(Initial Sync)は新しいメンバーにデータセット全体を表示するためのもので、レプリケーションはデータセット全体に継続的な変更を適用するためのもの。

Syncプロセス

- oplog には、プライマリが実行するすべての操作が含まれている
- 各メンバーのローカルデータベースのCappedコレクション(サイズ制限がついたコレクション)
    - 一定量貯まると消える
- どのメンバーもレプリケーションのソースとして使用できる
- Syncのステップ (例：targetコレクションに書き込み発生)
    - targetコレクションにデータ書き込み
    - primaryサーバー内でデータ操作、登録日時などをoplogに書き込む
    - secondaryがprimaryのoplogを非同期に確認 登録日時から差分をチェック
        - 差分があれば、secondaryのoplogに差分が書き込まれる
    - secondaryのoplogから、targetコレクションにデータ操作が反映される
- 同じオプログ操作を再度適用した場合、同じ結果になる
- oplogは固定サイズ (固定時間ではない)
    - 通常 1 つのデータ操作はoplog上の 1 つの操作になる
    - 例外: 複数のドキュメントに影響を与える一つの操作は、単一のドキュメントに対する多くの操作として oplog に格納される

- セカンダリは次のような場合に古くなる可能性がある:
  - セカンダリ がダウンしている場合
  - 処理性能よりも多くの書き込みが発生した場合
  - 読み込み処理のためビジーになっている場合
- oplogのサイジングはメンテナンス時間を確保するために重要

イニシアルSyncプロセス

- 初期同期は、レプリカセットの一つのメンバーから別のメンバーにすべてのデータをコピーする。
- 初期同期時、ステータスはSTARTUP2となる
- ローカルデータベース以外のすべてのデータベースをクローンする。
  - クローンを作成するために、mongod は各ソースデータベースのすべてのコレクションをスキャンし、すべてのデータをそれらのコレクションの独自のコピーに挿入する。
- 各コレクションのドキュメントがコピーされると同時にすべてのコレクションのインデックスが構築される。
- データコピー中に新しく追加された oplog レコードがプルされる。
- データ セットにすべての変更を適用する。ソースからのoplogを使用して、mongodはレプリカセットの現在の状態を反映するようにデータセットを更新する。
- 初期同期が終了すると、メンバーの状態はSTARTUP2からSECONDARYに遷移する
- セカンダリーのメンバー 1 つを停止し、データベースディレクトリを空にしてメンバーを起動すると、手動で初期同期を実行できる。

レプリカセットメンバーの状態

- Startup
  - 初めてメンバーになった時。
- Secondary
  - Primaryからデータをレプリケーティングしているメンバー。投票可能。
- Primary
  - 書き込み可能な唯一のメンバ。投票可能

- Startup2
  - Initial Sync中の状態。レプリケーションと選挙処理を開始した状態。
- Recovering
  - エラー状態もしくは読み取りを受け入れる準備ができていない状態、多くの状況で発生
- Arbiter
  - 投票に参加するためだけに存在するメンバー
- Down
  - 連絡が取れなくなったメンバー
- Unknown
  - 他のメンバーから知られていない、一度も連絡が取れていないメンバー
- Removed
  - レプリカセットから削除された状態。この後再び追加すると "Normal "の状態に戻る
- Rollback
  - データをロールバックしているメンバー。読み取り不可。投票可能。

ロールバック

1. ロールバックとは何か、いつ実行されるか
   1. 選挙でレプリカセットがプライマリを置き換えるときは、古いプライマリにはセカンダリメンバーに複製されなかったドキュメントが含まれている可能性がある。この場合、古いプライマリ・メンバーはそれらの書き込みを戻す(ロールバック)。ロールバックの間、そのメンバーはROLLBACKというステータスになる。
   2. ロールバックは、フェイルオーバー後にメンバーがレプリカセットに再参加したときに、以前のプライマリの書き込み操作を元に戻します。ロールバックが必要なのは、プライマリがステップダウンする前にセカンダリが正常にレプリケートできなかった書き込み操作をプライマリが受け付けていた場合のみ。プライマリがセカンダリになってセットに再参加したときは、他のメンバーとのデータベースの整合性を保つために書き込み操作を元に戻したり「ロールバック」したりする。
   3. 。ロールバックが発生するのは、ネットワークの分断が原因であることが多い。セカンダリが前のプライマリに対する操作のスループットに追いつけなくなると、ロールバックの規模や影響が大きくなる。

      ※ プライマリがステップダウンする前に、レプリカセットの別のメンバーに書き込み操作がレプリケートされ、そのメンバーが利用可能で、レプリカセットの過半数がアクセス可能な状態が維持されていれば、ロールバックは発生しない。

2. ロールバックされたデータの同期が必要な場合、手動で行う必要がある
    - データディレクトリ内のrollbackディレクトリにある<database>.<collection>.<timestamp>.bsonファイル にロールバックデータが保存される(MongoDBのバージョンにより保存場所に差異あり)
    - 上記のファイルを一時的なコレクションにmongorestoreし、手動でマージを実行する
3. 300MB以上のデータがある場合は自動ロールバックに失敗する(ver 4.0以降はサイズの制限なし)
4. ロールバックを防ぐ方法
    1. ジャーナリングを有効にする
    2. writeConcern でw: majorityを使用する
    3. セカンダリーを最新の状態に保つ

アプリケーションからの接続 / ドライバ

- アプリケーションからMongoDBに接続する際、接続文字列を使用して接続する場合がある。
- 接続文字列の例: mongodb://user_name:password@host1:27017,host2:27018,host3:27019/?replicaSet=replicaName&connectTimeoutMS=10000&authMechanism=SCRAM-SHA-1
    - authMechanism・・認証メカニズムの指定
- Driver
    - プログラム言語を用いて接続するにはドライバが必要。
    - Java、Python、Node.js、Rubyなど、主要なプログラム言語のドライバが公式から提供されている。
    - Driverのドキュメント： https://docs.mongodb.com/drivers/

書き込み確認 writeConcern

- 書き込み確認は、データの登録・更新・削除の際に、どこまで書き込みしたかをもって完了と判断するかを設定できる
- MongoDBの書き込み時の挙動
    - insertなどの書き込みをした際、その内容はDBサーバーのメモリに保持される。その内容を0.05秒ごとにジャーナルファイルに書き込みをしている
- 書き込み確認のオプション
    - w・・・書き込みの台数を指定
        - w=0 書き込み確認を行わない
        - w=1 デフォルト設定。Primaryへの書き込みを確認

        - w=N プライマリに書き込まれ、そしてそれが N-1 のセカンダリにレプリケートされたことを確認
        - w=majority メンバーの過半数に書き込みがされることを確認
    - j・・・trueの場合、ジャーナルに書き込みされるまで確認
        - j=false デフォルト設定。メモリまで書き込みされることを確認
        - j=true データがジャーナルファイルに書き込まれるまでことを確認
    - wtimeout 書き込み完了までのタイムアウト時間(ミリ秒)
- 書き込み確認するメンバが多ければ多いほど、プライマリが失敗した場合に書き込まれたデータがロールバックする可能性は低くなる。しかし、高い書き込み確認を指定すると、クライアントは要求されたレベルの書き込み確認を受け取るまで待たなければならないため、待ち時間が長くなる可能性がある。

wにmajorityを指定した場合、投票権を持つメンバーの過半数に書き込みが完了したら書き込み成功
> db.testColl.insert({a: 1}, {writeConcern: {w: "majority"}})
> db.testColl.insert({a: 1}, {writeConcern: {w: "majority", wtimeout: 5000}})

wに数字を指定した場合、その台数分のReplica Setメンバーへの書き込みが完了したら書き込み成功と判断
> db.testColl.insert({a: 1}, {writeConcern: {w: 3, wtimeout: 5000}})
writeConcernのエラー例。

np_rep:PRIMARY> db.testColl.insert({a: 1}, {writeConcern: {w: 5, wtimeout: 5000}})
WriteResult({
"nInserted" : 1,
"writeConcernError" : {
"code" : 64,
"codeName" : "WriteConcernFailed",
"errInfo" : {
"wtimeout" : true
},
"errmsg" : "waiting for replication timed out"
}
})

書き込み確認のデフォルト設定は、getLastErrorDefaults で指定する

以下の例は、w:過半数、タイムアウト:5000をデフォルトとして設定している
```
> cfg = rs.conf()
> cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout: 5000 }
> rs.reconfig(cfg)
```

タグとgetLastErrorModesという設定を組み合わせると、独自の書き込み確認を定義できる。以下に例を挙げる。

レプリカセットメンバーにタグを追加する
```
> config = rs.config()
> config.members[0].tags = {"dc": "FL"}
> config.members[1].tags = {"dc": "US"}
> config.members[2].tags = {"dc": "JP"}
> rs.reconfig(config)
```

カスタムのwriteConcernの定義
```
> config = rs.config()
> config.settings.getLastErrorModes = {
"multiDataCenters" : {"dc": 2},
"allDataCenters" : {"dc": 3}
}
> rs.reconfig(config)
```

カスタマイズされた書込保障「multiDataCenters」を使用
```
> db.testColl.insert({a: 1}, {writeConcern: {w: "multiDataCenters", wtimeout: 5000}})
```

Read Preference

- read preference は、クライアントからMongoDBの読み取りを行う際、どのメンバーに対してアクセスを行うかを指定できる設定
- read preferenceモード一覧：

    - primary
        - デフォルトの設定
        - プライマリからのみ読み取りを行う
    - primaryPreferred
        - 基本的にプライマリから読み取るが、処理が行えない場合にセカンダリから行う
    - secondary
        - セカンダリのみから読み取りを行う
    - secondaryPreferred
        - 基本的にセカンダリから読み取るが、処理が行えない場合プライマリから行う
    - nearest
        - ネットワーク遅延が最も低いメンバーから読み取りを行う
- タグセットを指定してread
    - read preferenceにtagSetオプションを指定すると、指定したタグセットを持つメンバーがから読み取りを行う。

プライマリに接続

$ mongo --host np_rep/localhost:27100,localhost:27101,localhost:27102

セカンダリを指定してクエリ発行 (cursor.readPref()を使用)

np_rep:PRIMARY> db.rp1.find().readPref('secondary')
2020-12-05T15:53:53.740+0900 I NETWORK [thread1] Successfully connected to localhost:27101 (2 connections now open to localhost:27101 with a 0 second timeout)
{ "_id" : ObjectId("5fcaf5e5506802c96a8868ea"), "x" : 1 }

explainでどのメンバーに接続するかを確認

np_rep:PRIMARY> db.rp1.find().readPref('secondary').explain()

（前後略）

"serverInfo" : {
"host" : "localhost.localdomain",
"port" : 27101,

"version" : "3.6.21",
"gitVersion" : "1cd2db51dce4b16f4bc97a75056269df0dc0bddb"
},

[タグセットを指定したread]

現在のタグセットを確認

np_rep:PRIMARY> rs.conf().members

dc=JPを指定してread

np_rep:PRIMARY> db.rp1.find({}).readPref("secondaryPreferred", [{ "dc": "JP"}])

[演習]

Read Preferenceを指定してクエリを発行してください。

レプリカセットの管理

スタンドアロン起動

- バックアップ時など、レプリカセットから切り離しスタンドアロンで起動したい場合、次の手順で行う。
    1. 対象のメンバーをシャットダウンする
    2. mongodを--replSetなしで、別のポートで、同じ--dbpathで実行する

mongod --shutdown --dbpath /data/np_rep/rep1
mongod --port 27999 --dbpath /data/np_rep/rep1

再設定の強制

- rs.reconfig()で設定を適用する際、forceオプションを使うと、セカンダリに設定を適用することができる
- 過半数のメンバーがダウンしている場合に有用
- 設定を正しく準備する必要がある
- forceはバージョンを劇的に増加させる

セカンダリに接続

versionの事前確認

> rs.conf()
"version" : 16,

設定を変更
> var config = rs.config()
> edit config // 接続したセカンダリのpriorityを上げてみましょう

セカンダリへのreconfigはforce=trueをしなければエラーとなる
> rs.reconfig(config)
reconfigをforce=trueによりセカンダリに適用する
> rs.reconfig(config, {"force" : true})

versionが大きくなることを確認

> rs.conf()

"version" : 92686,

メンバーの状態の手動変更

- メンバーのpriorityを変更することで、どのメンバーをプライマリにするかをコントロールすることができる
  - より高いpriorityでそのメンバーをプライマリにする
- プライマリをセカンダリに降格させる。指定した期間(秒)はプライマリに戻らない (デフォルトでは60秒)
  - rs.stepDown(<秒数>)

プライマリに接続
$ mongo --host np_rep/localhost:27100,localhost:27101,localhost:27102

プライオリティを変更
> cfg = rs.conf()
> cfg.members[0].priority = 0.5
> cfg.members[1].priority = 0.5
> cfg.members[2].priority = 1
> rs.reconfig(cfg)

どのメンバーがプライマリかを確認
rs.isMaster()

> rs.stepDown() > rs.stepDown(600)
[演習] primaryに接続してセカンダリに降格させ、どのメンバーがprimaryになるかを確認してみましょう。

選任防止

- rs.freeze(<ミリ秒>)で、指定したミリ秒数、primaryになるのを防ぐ。rs.freeze()で解除。

- rs.stepDownとrs.freezeを組み合わせることで、任意のセカンダリをプライマリに昇格させることができる

> rs.freeze(600)

メンテナンスモード

- サーバが長時間の処理を行っている場合やレプリケーションが遅れている場合に有用
- セカンダリに対して行う。プライマリには不可。
- RECOVERING状態
    - データの読み取り不可。
    - プライマリから同期は続行。

セカンダリに接続し、RECOVERING状態にする
> db.adminCommand({"replSetMaintenance" : true})

状態確認
rs.status()
(前後略)
    "state" : 3,
    "stateStr" : "RECOVERING",

通常に戻す
> db.adminCommand({"replSetMaintenance" : false})

[演習]セカンダリに接続し、replSetMaintenanceを使用してRECOVERING状態にしてください。データベース読み込みができないことを確認してください。その後、通常の状態にもどしてください。

レプリケーションの監視

- status of the replica set (from the current server perspective)
- db.adminCommand("replSetGetStatus") or rs.status()

- important fields: self, stateStr, uptime [s], optimeDate (last oplog operation), pingMs, errmsg
- レプリカセットの状態
- db.adminCommand("replSetGetStatus") または rs.status()
- 重要なフィールド:
    - self: 現在このコマンドを実行しているインスタンスのメンバーであればtrueとして表示される
    - stateStr: 状態を表示。PRIMARY, SECONDARYなど。
    - uptime [s]: オンラインになってからの秒数。
    - optimeDate: 最後にoplogを操作した日時。 lastHeartbeatと比較し非常に古い場合、遅延が発生していると考えられる。
    - pingMs: localからリモートへの往復時間をミリ秒で表示。rs.status()を実行したメンバーはこの項目は表示されない

> rs.status()

[演習]

rs.status()を実行し、各メンバーのoplog に遅延が発生していないかを調べてください

レプリケーションソース

- rs.status() を使用してレプリケーション元を確認することができる
    - syncSourceHostに各メンバーの同期元のホストを表示
- レプリケーションソースは 最小のping時間で決定される。
    - セカンダリは、セットのデータの最新のコピーを維持するために、プライマリ・メンバーからデータをキャプチャする。ただし、デフォルトでは、セカンダリは、メンバー間のping時間や他のメンバーのレプリケーションの状態の変化に基づいて、セカンダリのメンバーへの同期先を自動的に変更する。
- レプリケーションでチェーンが生成される可能性がある。
    - MongoDB はChains(連鎖)レプリケーションをサポートしている。連鎖レプリケーションとは、セカンダリのメンバーがプライマリからではなく別のセカンダリからレプリケートすること。たとえば、セカンダリが ping の時間を基準にレプリケーション先を選択していて、一番近いメンバーが別のセカンダリだった場合など。
    - チェーンレプリケーションはプライマリの負荷を減らすことができる。しかし、チェーンレプリケーションはネットワークのトポロジーによっては、レプリケーションのラグを増大させる結果になることもある。
    - 連鎖レプリケーションがラグの原因になっている場合は、 Replica Set Configuration の settings.chainingAllowed 設定を使って連鎖レプリケーションを無効にすることができる。

replSetSyncFromはセカンダリーの現在のデフォルトの同期元を一時的にオーバーライドする
np_rep:SECONDARY> db.adminCommand({"replSetSyncFrom" : "localhost:27102"})

同期元の確認
rs.status()

MongoDB はデフォルトでChains(連鎖)レプリケーションを有効にしている。この手順では、下記は無効にする方法。
> var config = rs.config()
> config.settings.chainingAllowed = false
> rs.reconfig(config)

設定の確認

> rs.config().settings

[演習] 連鎖レプリケーションを変更してみましょう。

Oplogのリサイズ

- Oplogのサイズは故障時の復旧可能時間に影響を与えるので、重要
- 余裕のあるサイズのoplogによりメンテナンスの時間を確保
    - oplogはcapped collection、容量に制限のあるcollectionである。
      つまり、例えばセカンダリが長時間のダウンでoplogが保存されている期間を過ぎた場合は、oplogによる同期が不可能になってしまう(stale状態)。
      こうなってしまうと、データを空にしInitial Syncではじめから同期するか、データを手動コピーする必要がある。
- プライマリのoplogは、少なくとも24時間から数日分のデータが入るように設定した方がよい

printReplicationInfoは、プライマリーノードから見えるレプリカセットの状態をレポートフォーマットで提供
メンバーのoplogのフォーマットされたレポートを出力する

> db.printReplicationInfo()
configured oplog size: 192MB <-- oplogのサイズ
log length start to end: 341306secs (94.81hrs) <-- oplogが現在記録しているデータの時間数
oplog first event time: Sun Nov 15 2020 13:37:04 GMT+0900 (JST)
oplog last event time: Thu Nov 19 2020 12:25:30 GMT+0900 (JST)
now: Thu Nov 19 2020 12:25:40 GMT+0900 (JST)
log length start to endは、oplogが現在記録しているデータの時間数を出力する。
たとえば、これが常に48時間以上であるとすると、Primaryの復旧にかける時間を48時間確保できるということです。

JSONフォーマットで出力
> db.getReplicationInfo()

リサイズ手順

- すべてのセカンダリに対して replSetResizeOplogコマンドを使用してoplogサイズを変更する。
- その後、プライマリに対してreplSetResizeOplogコマンドを使用してoplogサイズを変更する。

セカンダリに接続

> mongo <hostname>:<port>

(Optional)現在のoplogサイズの表示

oplogの現在のサイズを表示するには、ローカルデータベースに切り替えて、oplog.rsコレクションに対してdb.collection.stats（）を実行する。 stats（）は、oplogサイズをmaxSizeとして表示する。

> use local
> db.oplog.rs.stats().maxSize

メンバーのoplogサイズ変更

oplogのサイズを変更するには、replSetResizeOplogコマンドを実行し、サイズパラメーターとしてメガバイト単位の目的のサイズを渡す。 指定するサイズは、990または990メガバイトより大きくする必要がある。

次の操作は、レプリカセットメンバーのoplogサイズを990メガバイトに変更する。

```
> db.adminCommand({replSetResizeOplog: 1, size: 990})
```

(Optional) oplogを縮小した場合、ディスクスペースを再利用するためにはcompactコマンドを行う。

```
> use local
> db.runCommand({ "compact" : "oplog.rs" } )
```

oplogサイズの確認

```
> db.oplog.rs.stats().maxSize
```

[演習]
すべてのセカンダリのoplogサイズをreplSetResizeOplogにより変更してください。

そのあと、プライマリのoplogサイズを変更してください。

運用中のレプリカセットに対するインデックス構築

- 運用中のレプリカセット上でインデックス構築する場合、ローリング方式でのインデックス構築が推奨される
    - ローリング方式でのインデックス構築をしない場合、インデックス作成時にread/writeにロックが発生する可能性がある
- ローリング方式でのインデックス構築手順
    - 一つのセカンダリを停止し、スタンドアロンで起動
        - そのセカンダリでインデックス作成
            - db.records.createIndex({ username: 1 })
    - そのセカンダリを停止し、レプリカセットのメンバーとして起動
    - 上記をセカンダリすべてに対して行う
- プライマリをステップダウンし、スタンドアロンとして起動
    - インデックス作成し、レプリカセットのメンバーとして起動

レプリケーションのLag(遅延)

- レプリケーションのlag(遅れ)とは、プライマリでの操作が oplog からセカンダリに適用されるまでの遅れのこと。

レプリケーションの遅れは重大な問題であり、 MongoDB のレプリカセットの導入に深刻な影響を及ぼします。過度のレプリケーション遅延は「遅延している」メンバーをすぐにプライマリにするのに不適格にしてしまいますし、分散型の読み込み操作が矛盾する可能性が高くなります。

- 現在のレプリケーションラグの長さを確認するには
    - プライマリに接続された mongo シェルで、rs.printSecondaryReplicationInfo() メソッドを呼び出す
      これは各メンバの syncedTo 値を返す。最後の oplog エントリがセカンダリに書き込まれた時刻を示している。

np_rep:PRIMARY> rs.printSecondaryReplicationInfo()
(実行例)
source: localhost:27100
syncedTo: Sat Dec 05 2020 18:34:07 GMT+0900 (JST)
10 secs (0 hrs) behind the primary
source: localhost:27101
syncedTo: Sat Dec 05 2020 18:34:07 GMT+0900 (JST)
10 secs (0 hrs) behind the primary
source: localhost:21705
no replication info, yet. State: (not reachable/healthy)
source: 127.0.0.1:27106
syncedTo: Thu Jan 01 1970 09:00:00 GMT+0900 (JST)
1607160857 secs (446433.57 hrs) behind the primary
source: 127.0.0.1:27107
syncedTo: Thu Jan 01 1970 09:00:00 GMT+0900 (JST)
1607160857 secs (446433.57 hrs) behind the primary

Lagの原因

lagの原因としてはいかのものが挙げられる

* ネットワーク遅延

セットのメンバー間のネットワークルートをチェックして、パケットロスやネットワークルーティングの問題がないことを確認します。

ping などのツールを使用してセットメンバー間のレイテンシをテストしたり、traceroute を使用してパケットのネットワークエンドポイントのルーティングを公開したりします。
* ディスクスループット

セカンダリのファイルシステムとディスクデバイスがプライマリと同じように素早くデータをディスクに流すことができない場合、セカンダリは状態を維持することが困難になります。ディスク関連の問題は、仮想化インスタンスを含むマルチテナントシステムでは非常に多く見られ、システムが IP ネットワークを介してディスクデバイスにアクセスすると一過性のものになることがあります（Amazon の EBS システムの場合のように）。

ディスクの状態を評価するには、iostat や vmstat などのシステムレベルのツールを使用します。

* Concurrency 並行処理

場合によっては、プライマリでの長時間の操作によってセカンダリへのレプリケーションがブロックされることがあります。最良の結果を得るためには、セカンダリへのレプリケーションの確認を要求するように write concern を設定してください。これにより、レプリケーションが書き込み負荷に追いついていない場合に書き込み操作が戻ってくるのを防ぎます。

また、データベース プロファイラを使用して、遅延の発生に対応する遅いクエリや長時間実行される操作がないかどうかを確認することもできます。

* 適切でないWrite Concern

プライマリへの大量の書き込みを必要とする大規模なデータインジェストやバルクロード操作を実行している場合、特に書き込み確認なしのWrite Concernがあると、セカンダリは変更に追いつくのに十分な速さで oplog を読み取ることができません。

これを防ぐには、セカンダリがプライマリに追いつくための機会を提供するために、100、1,000、またはその他の間隔ごとにwriteConcernを要求してください。