

MongoDB for Developer

IGS / ACD_RD7
duyhsieh
duyhsieh@igs.com.tw

source file available at:
<https://github.com/mantisa1980/pythontutorial>

Agenda

- NoSQL簡介
- MongoDB
 - mongo shell
 - query & index
 - pymongo CRUD
 - Replica set
 - Sharding
- 課程總結: 簡易帳號查詢系統實作

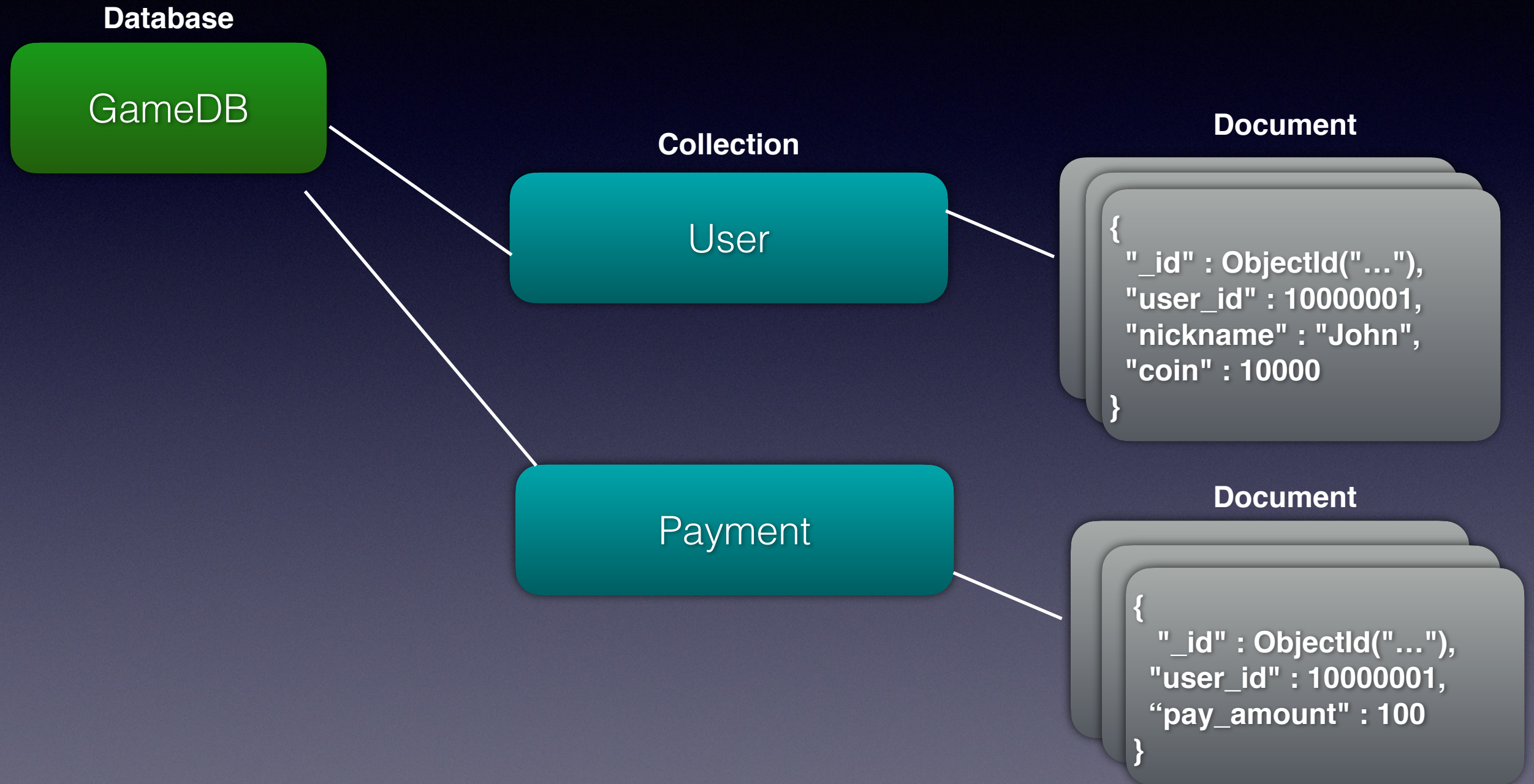
NoSQL: Not only SQL

- NoSQL: 2009年後開源社群對新興分散-非關聯式資料庫統稱
- 主流NoSQL類型
 - Key-value database: Google BigTable, Hadoop HBase, Cassandra
 - In-memory database: Memcached, Redis
 - Document database: MongoDB, CouchDB
 - Graph database: Neo4j

About MongoDB

- Document database
 - Database: 包含一群collection
 - Collection: 包含一群documents
 - Document (Record): json data record
 - 實際儲存的是BSON: binary-encoded json structure
- 可替換的Storage engine: V3.0預設為WireTiger
 - V3.0以前: MMAPV1

MongoDB Document Database Model



MongoDB優點

- 寫入/讀取速度比傳統RDBMS快
- 容易水平擴充
 - 傳統資料庫較依賴垂直擴充
- 無強制schema限制: loose schema
 - 對於處理非結構化/半結構化的資料較方便
 - Note: 不要濫用loose schema做出難以維護的DB

MongoDB缺點

- 不支援join查詢 (普遍NoSQL特性)
 - 通常是由AP層手動作多次查詢
 - 雖然3.2以後有lookup 指令，但join先天非mongoDB強項故有限制 (ex. 無法用在分散式資料庫上)
- 4.0以前不支援Transaction: 交易失敗無法rollback，也無跨collection寫入指令
 - 但對於single document 寫入是atomic: 若非成功，則必沒有寫入，不會只有寫一半
 - Mongo 4.0版提供ACID transaction，4.4版本後較穩定
- 做資料分析用傳統SQL較方便

Mongo Shell

- MongoDB 互動式介面
 - 常用來做簡單的操作
 - 為避免人為key錯，複雜的流程建議寫pymongo程式處理
- 基本指令練習: (tab for hints)
 - mongo 或mongo --port 27017 (連線: 預設主機127.0.0.1)
 - use DB1
 - db.Col1.insert({"name":"Hello"})
db.Col1.insert({"name":"World"})
 - db.Col1.find({"name":"Hello"})

pymongo

- Python MongoDB client package
- 大部分mongo shell指令都可以用pymongo做
- 2.x和3.x介面不相容，若已用2.x開發請勿升級到3.x
 - Note: MongoDB 3.x 可相容pymongo 2.x / 3.x API
- 底下範例皆以pymongo 2.8為準
 - 安裝pymongo 2.8: `pip install pymongo==2.8`

MongoClient物件

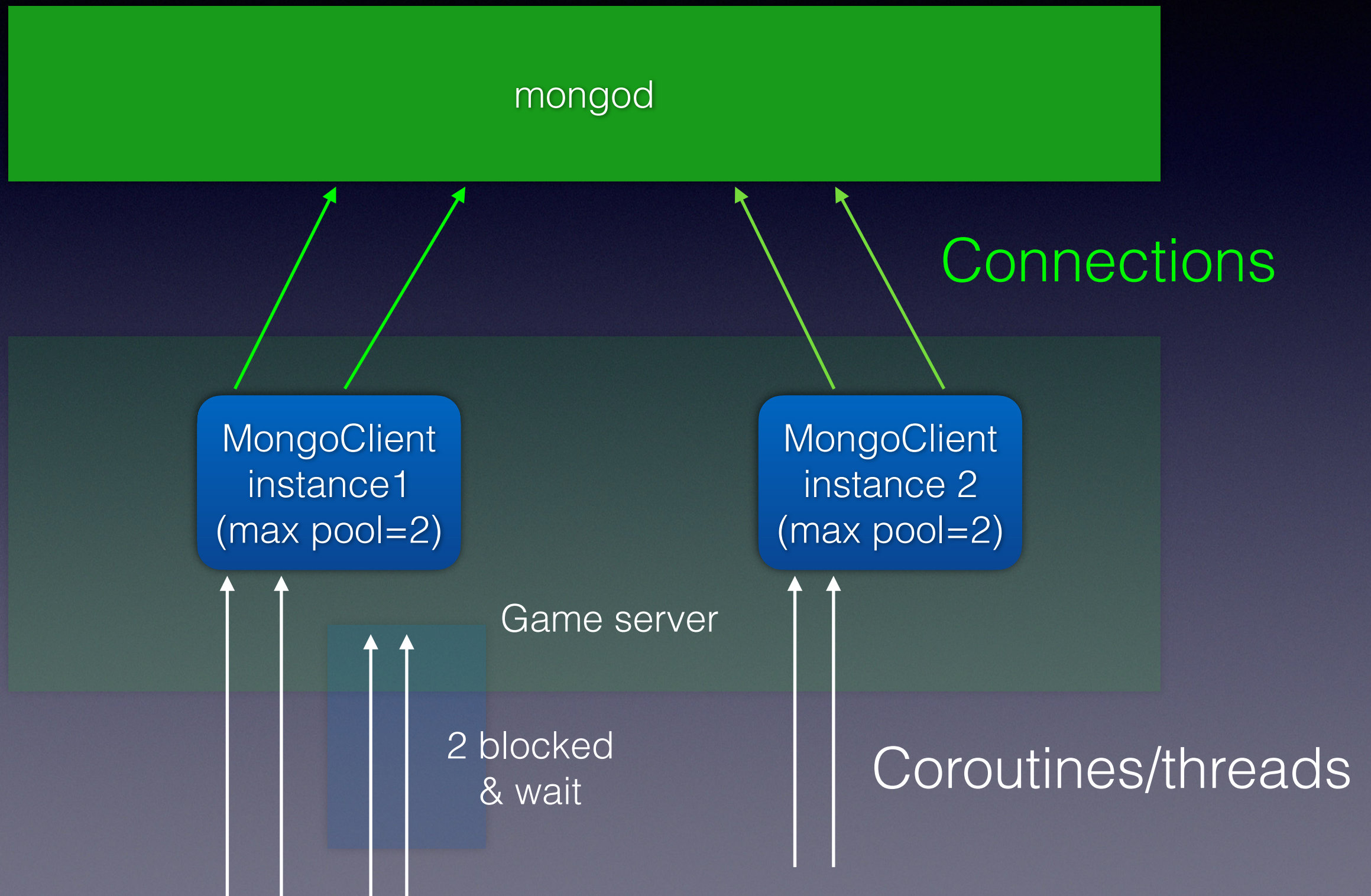
- Import pymongo
mc = pymongo.MongoClient(host="127.0.0.1")
- 取得名為User的database
db = mc["User"]
- 取得User db內的Wallet collection
col = db["Wallet"]
- 直接看Code練習：[mongo_test1.py](#)

A More Complex Connection

Example from Production

- `mc = pymongo.MongoClient(host="127.0.0.1", port=27017, max_pool_size=10, socketTimeoutMS=3000, connectTimeoutMS=1000, waitQueueTimeoutMS=3000, waitQueueMultiple=4)`
 - ***max_pool_size***: 該instance最大連線數
 - ***connectTimeoutMS***: 建立mongoDB等待回應的最長時間; 超過則觸發error
 - ***socketTimeoutMS***: 資料傳輸回應的最長等待時間; 超過則觸發error
 - ***waitQueueTimeoutMS***: 最長等待可用連線的時間; 超過則觸發error
 - ***waitQueueMultiple***: 最多可等待的請求者數目 (`max_pool_size * waitQueueMultiple`). 超過則不等待直接觸發error

Connection Pooling



CRUD

- Insert:

```
doc = {"user_id": "100000001", "coin": 1234 }  
col.insert(doc)
```

- 把docs放在list中insert，pymongo會自動轉為bulk insert

- Bulk insert: 把多筆資料一次一起送出去(pipeline)，資料量大時可以節省round trip time

- ```
doc = [{"user_id": "100000001", "coin": 1234 },
 {"user_id": "100000002", "coin": 5678 },
]
col.insert(doc)
```

- 練習：**mongo\_test2.py**



# CRUD

- 搜尋全部資料:  
`cursor = col.find()`
- 搜尋特定條件:  
`cursor = col.find( { "user_id":"10000001" } )`  
  
`cursor = col.find( { "user_id":"10000001",  
                    "coin": {"$gte":0 }  
                    })`
- `cursor = col.find({"user_id":{"$in":["10000001","10000002","10000003"]} })`
- comparison operators: `$eq`, `$gt`, `$lt`, `$gte`, `$lte`, `$ne`, `$in`, `$nin`...



# CRUD

- 讀取出cursor全部的資料:

```
cursor = col.find()
for document in cursor:
 print document
```

- cursor實作了python generator介面: Iterable (類似 C++ iterator)
- generator可以用for尋訪全部的元素



# CRUD: Aggregate

- Group syntax:
  - { \$group: { **\_id**:<expression>, <field1>: { <accumulator1>: <expression1> }, ... } }
  - **\_id**為Group Key(s)，不能省略；用\$取得要group的keys
- 假設collection內的資料為
  - {"store\_id": "A", "item\_id": 1, "capacity": 50 }, {"store\_id": "A", "item\_id": 2, "capacity": 100 }, {"store\_id": "B", "item\_id": 1, "capacity": 20 }, {"store\_id": "B", "item\_id": 3, "capacity": 40 }, {"store\_id": "C", "item\_id": 1, "capacity": 50 },

```
query = [{ "$match": { "store_id": { "$in": ["A", "B"] } } },
 { "$group": { "_id": "$store_id", "total": { "$sum": "$capacity" } } }]
ret = col.aggregate(query, cursor={ }) # note: mongo 3.6強制要給cursor參數(舊版直接傳Array全部回來)
for i in ret:
 print i
Result:
{'total': 60.0, '_id': 'B'}
{'total': 150.0, '_id': 'A'}
```



# CRUD: Aggregate

Group by 多欄位: `_id`後面接object

```
query = [{"$match": {} },
 {"$group": {"_id": {"STORE": "$store_id", "ITEM": "$item_id" }, "Total":
{"$sum": "$capacity"} } }]
ret = col.aggregate(query, cursor={ })
for i in ret:
 print i
```

Result:

```
{'total': 40.0, '_id': {'ITEM': 3.0, 'STORE': 'B'}}
{'total': 20.0, '_id': {'ITEM': 1.0, 'STORE': 'B'}}
{'total': 100.0, '_id': {'ITEM': 2.0, 'STORE': 'A'}}
{'total': 50.0, '_id': {'ITEM': 1.0, 'STORE': 'A'}}
{'total': 30.0, '_id': {'ITEM': 1.0, 'STORE': 'C'}}
```

• 練習 : *aggregate\_test.py*



# CRUD

- `db.col.update(<query>, <update>, <options>)`: 可update多個document
  - 也可以用`find_and_modify` (只會更新第一個找到的document)
- `col.update(  
 {"user_id": "100000001"},  
 {"$set": {"coin": 0}},  
 upsert=True)`
- `col.update(  
 {"user_id": "100000001"},  
 {"$inc": {"coin": 10000}},  
 upsert=True)`
- `$set / $inc` 可以包辦大部份update應用



# CRUD

- document可以有array欄位  
`col.insert({"name":1, "data":[1,2,3,4,5] })`
- Array建議只放固定設定檔，不要做array搜尋
  - 雖然有pop/push等操作也能建index，但mongodb query是針對document model設計，array使用上有些限制，query也不直覺
  - array塞太多東西撈出時難以檢視 (讀出一定是全讀，無法過濾)
- 練習:**mongo\_test3.py**
  - **Try fixing empty document problem**



# Atomic Operation

- 又稱隔離性: 一個指令執行期間不會有其他指令介入
  - Wire Tiger engine=document level lock
    - 寫入同一document指令為atomic，需排隊
  - MMAPV1為Collection-level lock



# Atomic Operation(Cont.)

- update / find\_and\_modify
  - single document: atomic
  - multiple document —> not atomic , update過程中可能有其他指令修改同一批documents
    - 若要atomic可用\$isolated功能，將整群document鎖住，但會使mongodb變成單執行緒執行且其他request需要等待，效能很差
    - 註: find\_and\_modify只會修改符合的第一筆document



# Atomic Update: 售票系統

- 如何確保票不超賣？

假設原始document為: {"id": "spiderman", "ticket\_count": 50 }

- ```
def buy_ticket():  
    doc = col.find_one({"id": "spiderman"})  
    if doc["ticket_count"] > 0:  
        col.update({"id": "spiderman"}, {$inc: {"ticket_count": -1}})  
        return True, doc["ticket_count"]  
    return False, None
```
- Any Problem ?

Atomic Update:售票系統(Cont.)

- 正解: 利用atomic指令，讓動作在DB內完成，不要將邏輯寫在 game server

- ```
r = col.find_and_modify({"id":"spiderman", "ticket_count":{"$gt":0} },
 {"$inc":{"ticket_count":-1} },
 fields={"_id":False}, new=False)
```

- if r != None:

- return True, r["ticket\_count"] # new=False:r為未inc之前的document

- return False, None

- 若要做json serialize，要去除\_id fields (ObjectId type無法序列化)



# Atomic Update(Cont.)

- 真實錯誤案例:產生玩家流水號ID
- 假設有個文件記錄目前流水號 : {"serial":1000}
  - ```
def create_player_id():  
    doc = col.find_one()  
    new_id = doc["serial"] + 1  
    col.update({}, {$inc:{"serial":1}})  
    return new_id
```
 - 2個玩家同時觸發request，可能會得到1001, 1001重複ID，造成災難
- 補充註解: 目前(2017/8/8) mongoDB有個issue: 對於find_and_modify搭配setOnInsert無法從find到insert都atomic，可能會重複insert到，或(有unique constraint) 觸發duplicate key error
 - issue: <https://jira.mongodb.org/browse/SERVER-14322?focusedCommentId=1130310&page=com.atlassian.jira.plugin.system.issuetabpanels%3Acomment-tabpanel#comment-1130310>

CRUD

- `col.remove({"user_id":"10000001"})` # 移除符合條件的全部document
- `col.remove({"user_id":"10000001"}, multi=False)` # 只移除符合條件的第一個document
- `col.remove({ })` # 移除全部document
- `col.drop()` # 移除整個collection (不會釋放空間，但空間會續用)
 - indexes也一併移除
- `mc.drop_database(db_name)` # 移除db_name DB，會釋放硬碟空間 (mc = mongo client instance)

Indexing

- 定義於Collection級別，非database
- 預設索引: `_id` (unique)
- 可建立ascending 遞增(1) 或descending遞減索引(-1)
 - (ascending) : $1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 4 \leftrightarrow 5$
 - (ascending, descending): $(1, 3) \leftrightarrow (1, 2) \leftrightarrow (1, 1) \leftrightarrow (2, 3) \leftrightarrow (2, 2) \leftrightarrow (2, 1)$
- 會佔用額外硬碟空間
- 更新document時要連同index一起更新
 - 不需要搜尋的database不要建索引.ex. dump backend logs

Indexing Types

- Example schema

```
{  
  "name": "John",  
  "age": 25,  
  "weight": 50  
}
```
- Single key example: {name:1}
- Compound key example: {name:1, age:-1}
- Others:
 - multi-key index(ex. a.b), hash index, geospatial index, text, index, partial index, sparse index

Query Sorting with Index

- 可依照想要的排序查詢資料，但必須符合index設定規則
 - 若不符合規則，index就無法發揮作用，造成搜尋效率低落
 - 若資料庫無法滿足排序條件，建議拉到game server排序，但須注意資料量IO
- 注意:若建立索引，讀出來的資料預設就是依照index排好的，不需要額外sort

Query Sorting with Index(Cont.)

- Syntax: `db.records.createIndex(sort_criteria)`
 - Single key: `db.records.createIndex({"a":1})`
 - Ex : 1 <-> 2 <-> 3 <-> 4 <-> 5
 - `sort_criteria: {"a":1}` (頭) / `{"a":-1}` (尾) : OK
 - Compound key: `db.records.createIndex({"a":1, "b":-1})`
 - Ex: (1, 3) <-> (1, 2) <-> (1, 1) <-> (2, 3) <-> (2, 2) <-> (2, 1)
 - `sort_criteria: {"a":1,"b":-1}` (頭), `{"a":-1,"b":1}` (尾): OK
 - `sort_criteria: {"a":1,"b":1}, {"a":-1,"b":-1}`: **not OK**

Covered Query

- 定義: 只使用index就能回傳結果，不需要從document取得資料，速度更快
 - 條件1: query全部fields都在對應index中
 - 條件2: query result return的field全都在對應index中
- ex. `db.inventory.createIndex({type:1, item:1})`
- `db.inventory.find(`
 `{ "type": "food", "item":"pork"}, // 符合條件1`
 `{ "item": 1, "_id": 0 }) // 排除_id field才符合條件2`

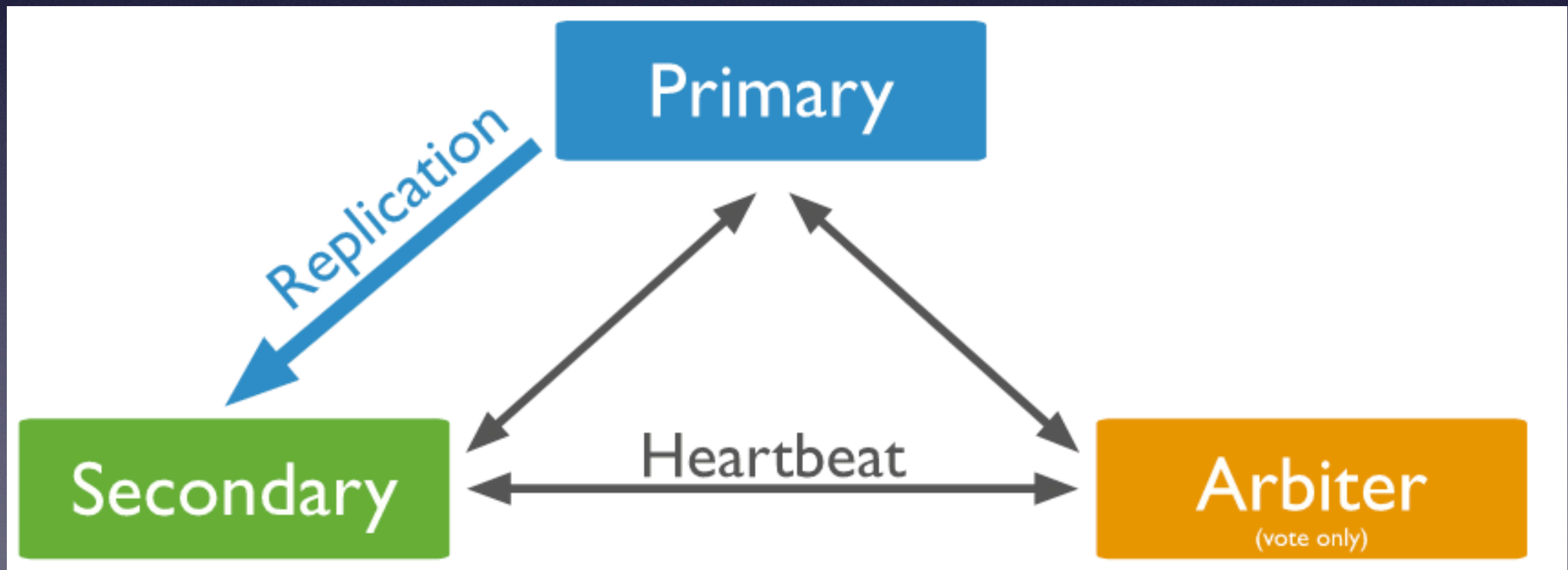
Error Recovery

- WireTiger每60秒做一次checkpoints (recovery point)更新
 - 但60秒中間的資料若中途shutdown必須靠journal
- Journal: sequential binary transaction log
 - 對資料庫的變更先寫入journal file
 - 寫入journal後的資料都可以recovery
 - 每隔一段時間再從journal file寫回disk的data file
- 關掉journal，mongoDB TPS會隨Core數增加，但不建議

Replica Set

- 備援機制
- 1 primary (RW)+ N secondary (R) + 1 arbiter
 - primary: 可讀寫
 - secondary: 可讀
 - Arbiter: 仲裁者，負責協調指派成員的primary/secondary角色. 消耗很少硬體資源

Replica Set(Cont.)



Replica Set特性

- 保持奇數成員: smooth election
 - 最小集合: 1 primary + 1 secondary+ 1 arbiter
- Secondary sync oplog from primary
 - oplog: high level database operations(insert/update..)
 - 屬於Statement-based replication (compared to row-based replication)
 - Journal: low-level disk operations

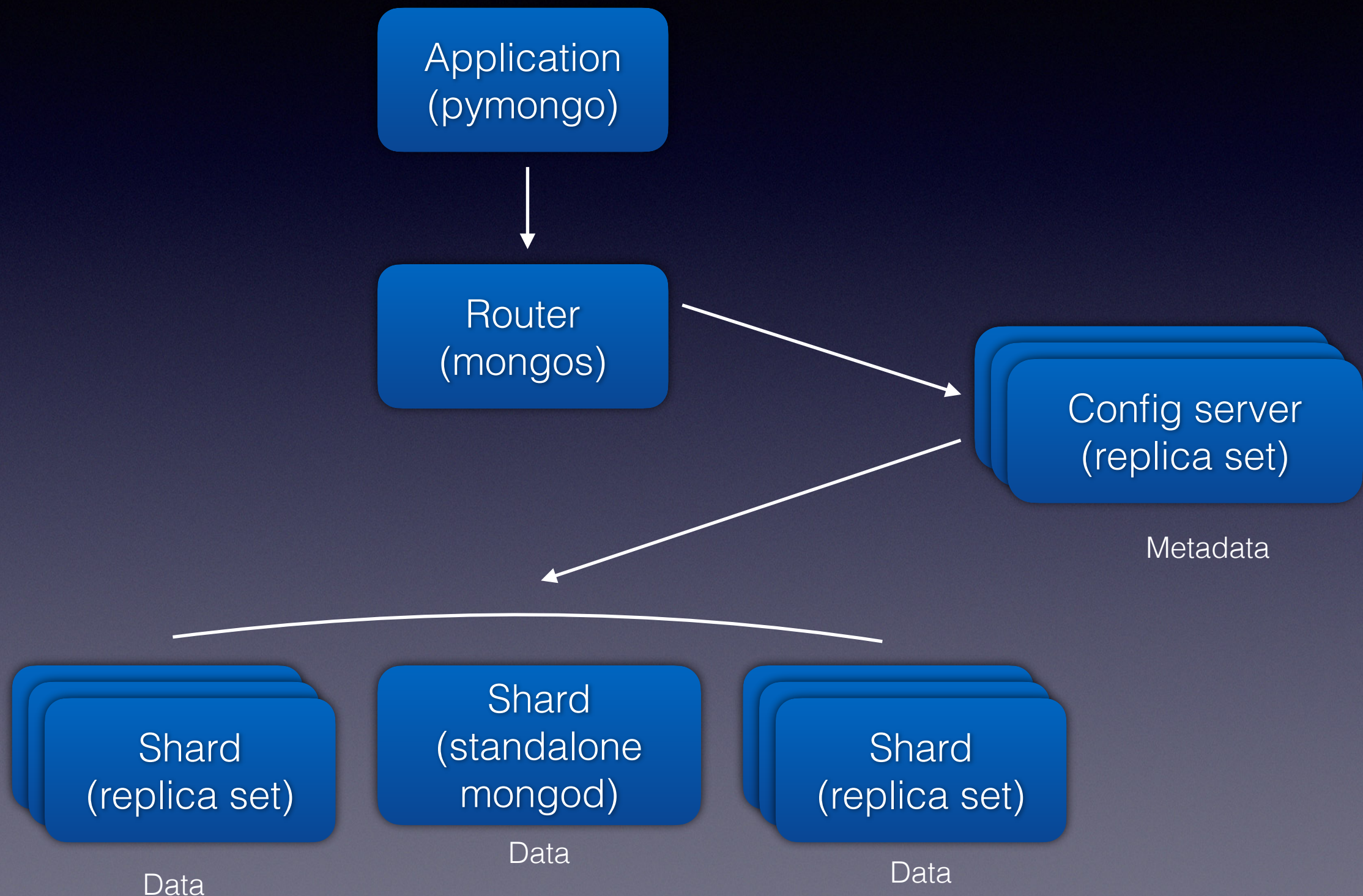
Replica Set特性((Cont.))

- eventually consistency: delayed update in secondaries
- 應用上常用replica set做讀寫分離，降低primary負載
 - 需注意secondary可能會讀到舊的 (eventually consistency)

Sharding

- 將資料切割到不同Shard上，分散負載
 - Shard(分片): 每個資料節點稱為一個shard
 - mongos透過config server同步shard metadata
- Chunk: Shard資料儲存單元，預設64MB
 - 當大於64MB會被分割成兩個chunks，再分散到其他shard上
- AP層透過mongos router存取；AP並不知道有分片機制
 - 不透過mongos直接進某shard查詢，只能查詢到部分資料

Shard(分片)



Shard Key

- 用來分配資料分佈的index
- 不可指定會變動的欄位：ex. coin / exp
- Shard key種類:
 - **range-based**: 根據key範圍做分割 ex. min~20000000->chunk1 , 20000001 ~ max -> chunk 2 ...
 - Note:流水號類型可能造成分佈不平衡
 - **hash-based**: 根據key的hash值做分配 partition. ex. 77->hash(77)-> chunk1 , 102->hash(102)->chunk2
 - Hash碰撞特性: 無法建立unique key , 不適用於玩家資料

建立Shard Key

- 注意: 需在mongos上操作
- `sh.enableSharding("UserDatabase")`
- `sh.shardCollection("UserDatabase.Account",
{"user_id":1 },
{ unique:true })`

課後練習

- src/answer資料夾下有一個簡易的server範例 (server.py)
- server跑起來連上首頁，web server會吐出一個簡易的jQuery網頁，方便操作CRUD功能
- 請修改程式碼新增新的web api或是mongoDB欄位 / Collection