COMP5338 – Advanced Data Models

Week 2: Document Store: Data Model and Simple Query

Dr. Ying Zhou
School of Information Technologies

THE UNIVERSITY OF SYDNEY

Administrative

- Lab arrangement

Time	Room	Capacity	Tutor
Tue 8-9pm	SIT114	30	Dai
Tue 8-9pm	SIT115	30	Andrian
Tue 8-9pm	SIT117	20	Heming (Taurus)
Tue 8-9pm	SIT118	20	Chenhao
Wed 5-6pm			
Wed 5-6pm			[illegible]urus)
Wed 5-6pm	*SIT118*	*20*	

- Most labs are not full at the moment
  - If you wish to move lab but cannot do it online, please go to the lab you want to attend and let the tutor know
- Students allocated in SIT118
  - If you wish to attend Wednesday labs, please attend SIT116 if your sid ends with even number and SIT117 if your sid ends with odd number
  - You may attend one of the Tuesday evening labs as well.

Outline

- **Overview of Document Databases**
- **MongoDB Data Model**
- **MongoDB CRUD Operations**

Structured and Unstructured Data

- Relational Database System is designed to store **structured data** in tabular format, e.g. each pieces of data is stored in a predefined field (attribute)

Supplier Table:

SuppID	Name	Phone
8703	Heinz	0293514287
8731		
8927		
9031	CSR	072097763

- **Unstructured data** does not follow any predefined "model" or "format" that is *aware to the underlying system* . Examples include data stored in various files, e.g word document

Semi-structured Data

- Many data have some structure but should not be constrained by a *predefined* and *rigid* schema
  - E.g. if some suppliers have *multiple* phone numbers, it is hard to capture such information in a relational model effectively
- **Self-describing** capability is the key characteristics of semi-structured data
  - schema/structur [illegible] ta, instead of a separate declaration
  - in database system, the structure [illegible] when you create a table. All rows need to follow the structure
  - in CSV and Excel, the structure is “declared" in the header row. All subsequent rows are supposed to follow that
- XML and JSON are two types of semi-structured data

A Self-describing XML document

```
<?xml version=“1.0” encoding=“UTF-8”?>
<invoice>
   <order-id> 1</order-id>
    <customer>
        <name> John</name>
        <address> Sydney</address>
    </customer>
  <products>
    <product>
        <code>123</code>
        <quantity>1</quantity>
    </product>
 </products>
</invoice>
```

metadata/structure information

data

Another invoice with slightly different structure

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice>
  <order-id> 2</order-id>
   <customer>
      <name> John</name>
      <address> Sydney</address>
      <contact>12345678</contact>
   </customer>
 <products>
   <product>
      <code>123</code>
      <quantity>1</quantity>
   </product>
  <product>
     <code>456</code>
     <quantity>2</quantity>
   </product>
 </products>
</invoice>
```

JSON Data Format

- JSON (**J**ava**S**cript **O**bject **N**otation) is a simple way to represent JavaScript objects as strings.
  - There are many tools to serialize objects in other programming language as JSON
- JSON was introduced in 1999 as an alternative to XML for data exchange.

- Each JSON obje st of property names and valu ces, in the following format:

```
{ propertyName1 : value1,           me2 : value2 }
```

- Arrays are represented in JSON with square brackets in the following format:

```
[ value1, value2, value3 ]
```

JSON format example

```
Invoice _1= {
            order-id: 1,
            customer: {name: “John”, address: “Sydney”},
             products:[ { code: “123”, quantity: 1}]
            }
```

```
Invoice _3= {
            order
            customer: {name: “Smith”,
                       address: “Melbou
                       contact: “12345”},
            products: [{ code: “123”, quantity: 20},
                        { code: “456”, quantity:2}]
            delivery: “express”
}
```

Document Databases

- Document database stores data in semi-structured documents
  - Document structure is flexible
- Provide own query syntax (different to standard SQL)
- Usually has pow
- Examples:
  - XML based database
  - JSON based database: MongoDB

Outline

- Overview of Document Databases
- **MongoDB Data Model**
- MongoDB CRUD operations

Matching Terms in SQL and MongoDB

SQL	MongoDB
Database	Database
Table	Collection
Index	
Row	ment
Column	
Primary key	
Join	Embedding and referencing $lookup in aggregation (since 3.2)

MongoDB Document Model

`users` table in RDBMS

Column name is part of schema

TFN	Name	Email	age
12345	Joe Smith	joe@gmail.com	30
54321	Mary Sharp	mary@gmail.com	27

two rows

Field name is part of data

Repeated in every document

```
{_id: 1
 name: “Joe Smith”,
 email: “joe@gmail.com”,
 age: 30
}
```

```
{ _id: 54321,
 name: “Mary Sharp”,
 email: “mary@gmail.com”,
 age: 27
}
```

two documents

`users` collection in MongoDB

Native Support for Array

```
{ _id: 12345,
  name: “Joe Smith”,
  emails: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30
}
```

```
{ _id: 54321,
  na
  em
  age
}
```

TFN	Name	Email	age
12345	Joe Smith	joe@gmail.com , joe@ibm.com ??	30
54321	Mary Sharp	mary@gmail.com	27

Native Support for Embedded Document

```
{ _id: 12345,
  name: “Joe Smith”,
  email: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30
}
```

```
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age:
  addr

          suburb: “chippenda
          zip: 2008
        }
}
```

TFN	Name	Email	age	address
12345	Joe Smith	joe@gmail.com	30	
54321	Mary Sharp	mary@gmail.com	27	1 cleveland street, chippendale, NSW 2008

MongoDB data types

- Primitive types
  - String, integer, boolean (true/false), double, null
- Predefined special types
  - Date, object id, binary data, regular expression, timestamp, and a few more
  - DB Drivers impl cific way
  - The interactive s r all
    - ISODate(“2012-09-11 18:00:00”)
- Array and object
- Field name is of **string** type with certain restrictions
  - “_id” is reserved for primary key
  - cannot start with “$”, cannot contain “.” or null

http://docs.mongodb.org/manual/reference/bson-types/

Data Modelling

- Key design decision in MongoDB data modelling involves how to represents *relationship* between data
  - How many collections should we use
  - What is the rough document structure in each collection
- Embedding or Referencing
  - Which object sh
    - And reference
  - Which object can be embedded in

http://www.mongodb.org/display/DOCS/Schema+Design

Referencing

- References store the relationships between data by including links or *references* from one document to another.

Embedding

- Embedded documents capture relationships between data by storing related data in a single document structure.

_id is not required

“Schema” Design Example

- A fully normalized relational model would have the following tables:
  - User
  -

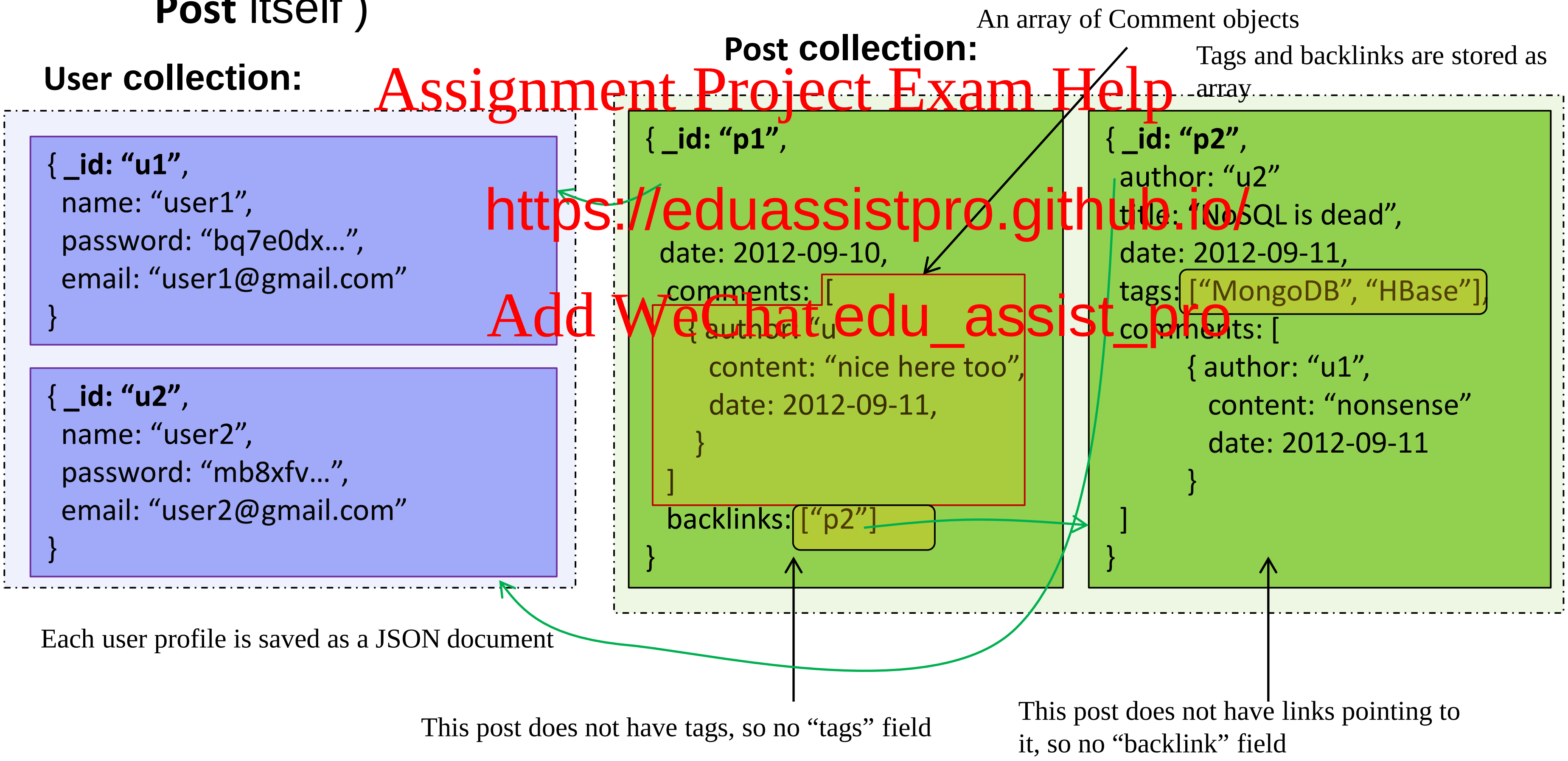
http://docs.mongodb.org/manual/applications/data-models/

MongoDB schema design

- Using **three** collections
  - **User** collection
  - **Post** collection (with links to **User**, **Comment**, and **Post** itself)
  - **Comment** Collection(with links to **User**)
- Using **two** collections
  - **User** collection
  - **Post** collection ([illegible] inks to **User** and **Post** itself

Two Collections Schema

- Two collections
  - **User** collection
  - **Post** collection (with *embedded* **Comment** object and links to **User** and **Post** itself)



User collection:

```
{ _id: "u1",
  name: "user1",
  password: "bq7e0dx…",
  email: "user1@gmail.com"
}
```

```
{ _id: "u2",
  name: "user2",
  password: "mb8xfv…",
  email: "user2@gmail.com"
}
```

Each user profile is saved as a JSON document

Post collection:

An array of Comment objects

Tags and backlinks are stored as array

```
{ _id: "p1",

  date: 2012-09-10,
  comments: [
   { author: "u
     content: "nice here too",
     date: 2012-09-11,
   }
  ]
  backlinks: ["p2"]
}
```

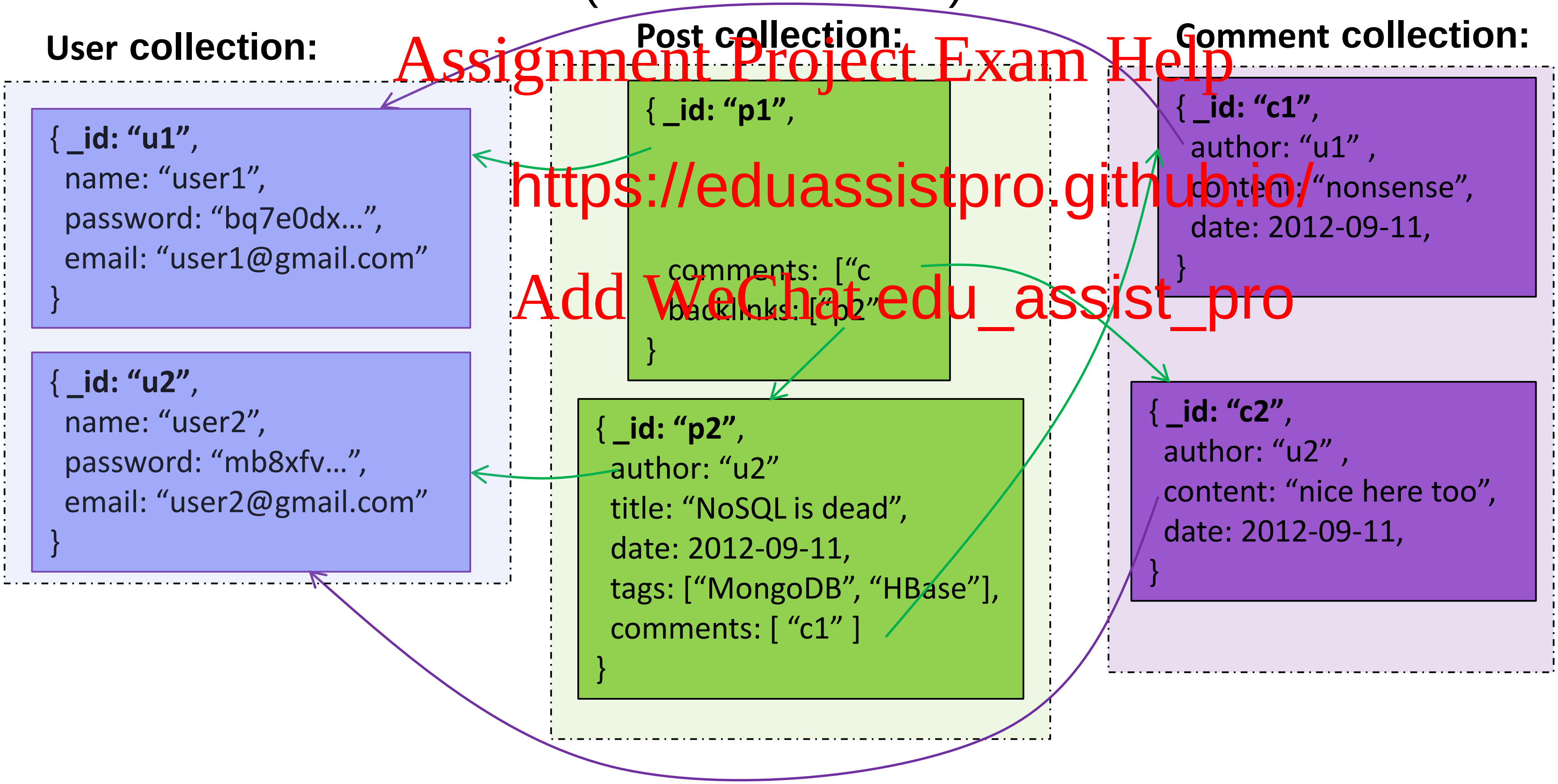
```
{ _id: "p2",
  author: "u2"
  title: "NoSQL is dead",
  date: 2012-09-11,
  tags: ["MongoDB", "HBase"],
  comments: [
     { author: "u1",
       content: "nonsense"
       date: 2012-09-11
     }
  ]
}
```

This post does not have tags, so no "tags" field

This post does not have links pointing to it, so no "backlink" field

Three Collections Schema

- Three collections
  - **User** collection
  - **Post** collection (with links to **User**, **Comment**, and **Post** itself)
  - **Comment** Collection(with links to **User**)



User collection:

```
{ _id: "u1",
  name: "user1",
  password: "bq7e0dx…",
  email: "user1@gmail.com"
}
```

```
{ _id: "u2",
  name: "user2",
  password: "mb8xfv…",
  email: "user2@gmail.com"
}
```

Post collection:

```
{ _id: "p1",


  comments: ["c
  backlinks: ["p2"
}
```

```
{ _id: "p2",
  author: "u2"
  title: "NoSQL is dead",
  date: 2012-09-11,
  tags: ["MongoDB", "HBase"],
  comments: [ "c1" ]
}
```

Comment collection:

```
{ _id: "c1",
  author: "u1" ,
  content: "nonsense",
  date: 2012-09-11,
}
```

```
{ _id: "c2",
  author: "u2" ,
  content: "nice here too",
  date: 2012-09-11,
}
```

Two Collections vs. Three Collections

- Which one is better?
  - Hard to tell by schema itself, we need to look at the actual application to understand
    - Typical data feature
      - What would happen if a post attracts lots of comments?
    - Typical queries
      - Do we want ost, or only the latest few, or not at all?
      - Do we need itself?
    - Atomicity consideration
      - Is there “all or nothing” update require to post and comment

- Other design variation?
  - In three collection schema, store post-comment link information in **Comment** collection instead of **Post** collection?
  - Embed the recent comments in **Post**?
  - One **User** collection with embedded **Post** and **Comment** objects? ✗
  - One **User** collection with **user**, **post** and **comment** documents? ✗

General Schema Design Guideline

- Depends on data and intended use cases
  - “independent” object should have its own collection
  - ***composition*** relationship are generally modelled as embedded relation
    - Eg. ShoppingOrder and LineItems, Polygon and Points belonging to it
  - ***aggregation*** relationship are generally modelled as links (references)
    - Eg. Department and Employee

  - **Many-to-Many** r ed as links (references)
    - Eg. Course an

  - If part-objects are always required w ect is queried, embed the part-object

    - We always want to display line items when displaying shopping order
    - We always want to display **Comments** along with the blog **Post**;
    - We always want to get Credit Card billing address when querying credit card information;
    - But we might not always want to get all students enrolled when querying about a course.

Course information page

Outline

- Overview of Document Databases
- MongoDB Data Model
- **MongoDB CRUD Operations**

MongoDB Queries

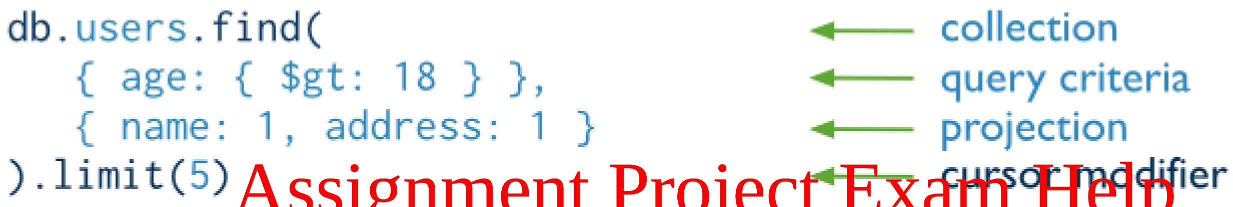
- In MongoDB, a ***read*** query targets a <u>specific</u> <u>collection</u>. It specifies **criteria**, and may include a **projection** to specify fields from the matching documents; it may include **modifier** to limit, skip, or sort the results.

- A ***write*** query m ete data. One query modifies the dat and delete query can specify query **criteri**

http://docs.mongodb.org/manual/core/crud-introduction/

Read Operation Interface

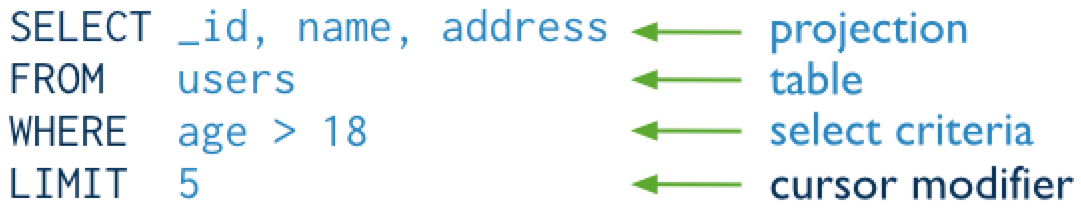
- db.collection.find()



```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

collection
query criteria
projection
cursor modifier

Find at most 5 d [illegible] ion with **age** field greater than 18, return only the na [illegible] ess field of each document.



```
SELECT _id, name, address
FROM   users
WHERE  age > 18
LIMIT  5
```

projection
table
select criteria
cursor modifier

Read Query Example

Find documents in the **users** collection with **age** field greater than 18, sort the results in ascending order by **age**

Read Query Features

- Users can find data using any criteria in MongoDB
  - Does not require indexing
  - Indexing can improve performance (week 4)
- Query **criteria** are expressed as BSON document (query object)
  - Individual condition is expressed using predefined selection operator, eg. **$gt** is the operator for "greater than"
- Query **projection** ument as well

SQL	**Mon n Shell**
select * from user	db.us ser.find({})
select name, age from user	db.user.find({},{name:1,age:1,_id:0})
select * from user where name = "Joe Smith"	db.user.find({name: "Joe Smith"})
select * from user where age < 30	db.user.find({age: {$lt:30}})

Querying Array field

- MongoDB provide various features for querying array field
  - https://docs.mongodb.com/manual/tutorial/query-arrays/
- The syntax are similar to querying simple type field
  - db.users.find({emails: “joe@gmail.com”})
    - Find user(s) whose email include “joe@gmail.com”.
  - db.users.find({“e
    - Find user(s) w om”.
  - db.users.find({emails: {$size:2}})
    - Find user(s) with 2 emails

```
{ _id: 12345,
  name: “Joe Smith”,
  emails: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30}
```

```
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27}
```

Querying Embedded Document

- Embedded Document can be queried as a **whole**, or by **individual field**, or by **combination of individual fields**
  - db.user.find({**address**: {number: 1, name: “pine street”, suburb: “chippendale”, zip: 2008}})
  - db.user.find({**“address.suburb”**: “chippendale”})
  - db.user.find({**“ad** **dress.suburb”**: “chippendale”})

```
{ _id: 12345,
  name: “Joe Smith”, email: [“joe@gmail.com”, “joe@ibm.com”], age: 30,
  address: {number: 1, name: “pine street”, suburb: “chippendale”, zip: 2008 }
}
```

```
{ _id: 54321,
  name: “Mary Sharp”, email: “mary@gmail.com”,age: 27,
  address: { number: 1, name: “cleveland street”,suburb: “chippendale”,zip: 2008 }
}
```

http://docs.mongodb.org/manual/tutorial/query-documents/#embedded-documents

Write Query- Insert Operation

Insert a new document in **users** collection.

Insert Example

- db.user.insertOne({**_id: 12345**, name: “Joe Smith”, emails: [“joe@gmail.com”, “joe@ibm.com”],age: 30})
- db.user.insertOne({ _id: 54321, name: “Mary Sharp”, email: “mary@gmail.com”, age: 27, address: { number: 1, name: “cleveland street”, suburb: “chippendale”, zi

user collection

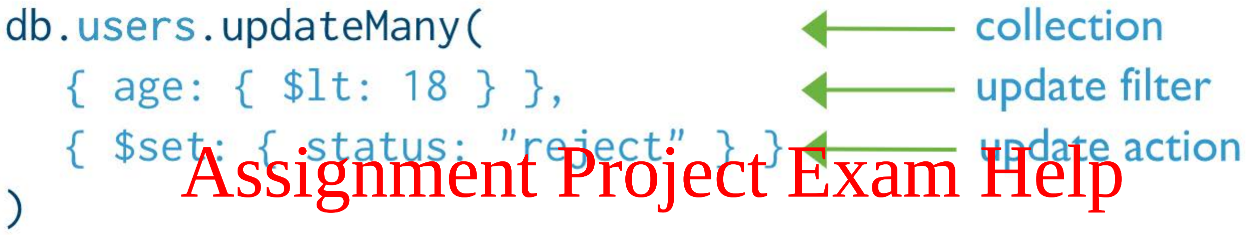
```
{ _
  emails: [“joe@gmail.co                m”],
age: 30
}
```

```
{ _id: 54321,
  name: “Mary Sharp”, email: “mary@gmail.com”, age: 27,
  address: { number: 1,
                   name: “cleveland street”,
                   suburb: “chippendale”,
                   zip: 2008
                 }
}
```

Insert Behavior

- If the collection does not exist, the operation will create one
- If the new document does not contain an “**_id**” field, the system will adds an “**_id**” field and assign a unique value to it.
- If the new docu **d**” field, it should have a unique v
- Two other operations:
  - **insertMany**
    - Insert many documents
  - **Insert**
    - Major language APIs only support **insertOne** and **insertMany**

Write Query – Update Operation



```
db.users.updateMany(
   { age: { $lt: 18 } },
   { $set: { status: ”reject” } }
)
```

Has the s[illegible] SQL:

Two other operations: **updateOne**, **replaceOne**

Updates operators

- Modifying simple field: $set, $unset
  - db.user.updateOne({_id: 12345}, {$set: {age: 29}})
  - db.user.updateOne({_id:54321}, {$unset: {email:1}}) // remove the field
- Modifying array elements: $push, $pull, $pullAll
  - db.user.updateOne({_id: 12345}, {$push: {emails: "joe@hotmail.com"}})
  - db.user.updateOne({_id: 54321},

    {$push: {emails: {$ @microsoft.com"]}}})
  - db.user.updateOne({_id: 12345}, {$p e@ibm.com"}})

{ **_id: 12345**, name: "Joe Smith", emails: ["joe@gmail.com", "joe@ibm.com"], age: 30}	{ **_id: 12345**, name: "Joe Smith", emails: ["joe@gmail.com", "joe@hotmail.com"], age: 29}
{ **_id: 54321**, name: "Mary Sharp", email: "mary@gmail.com", age: 27}	{ **_id: 54321**, name: "Mary Sharp", emails: ["mary@gmail.com", "mary@microsoft.com"] age: 27}

http://www.mongodb.org/display/DOCS/Updating

Write Operation - Delete

- db.user.deleteMany();
  - Remove all documents in user collection
- db.user.deleteMany({age: {$gt:18}})
  - Remove all documents matching a certain condition
- db.user.deleteOne({_id: 12345})
  - Remove one do condition

Isolation of write operation

- The modification of a single document is always **atomic**
  - It does not leave a document as partially updated.
  - A concurrent read will not see a partially updated document
  - This is true even if the operation modifies multiple embedded documents *within* a single document

- Read Uncommitt

  - Concurrent read nt that has been updated but not yet committed, or

  - If a write operation is subsequentl a concurrent read may return the updated value before it is rolled back

Single Document Atomicity

```
db.inventory.insertMany( [
  { item: "canvas", qty: 100, size: { h: 28, w: 35.5, uom: "cm" }, status: "A" },
  { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
  { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" }]);
```

```
db.inventory.updateO
   { item: "paper" },
   { $set: { "size.uom": "
   }
)
```

```
                , qty: 100,
   size: { h: 8.5, w: 11, uom: "in" },
   status: "D" }]);
```

```
db.inventory.find({item: "paper"})
```

```
{ item: "paper", qty: 100,
   size: { h: 8.5, w: 11, uom: “cm" },
   status: “D" }]);
```

```
{ item: "paper", qty: 100,
   size: { h: 8.5, w: 11, uom: “cm" },
   status: “P" }]);
```

Isolation of write operation

- If a write operation modifies multiple documents (**insertMany**, **updateMany**, **deleteMany**), the operation as a whole is not atomic, and other operations may interleave.
- Multi-Document Transactions is supported in version 4.0
- Other mechanis [illegible] versions
  - The **$isolated** op[illegible]eration that affects multiple docume[illegible]ds or writes once the first document is written
- All those mechanisms have gre[illegible]ance impact and are recommended to avoid if possible, document embedding is recommended as an alternative

Write Operation – interleaving Scenario

A write query comes

```
db.users.updateMany(
  { age: { $gt: 18 } },
  { $set: { status:  “A” } }
)
```

{age: 21, status: “U”}
{age: 23, status: “S”}
{age: 17, status: “E”}
{age: 25, status: “R”}
{age: 15, status: “S”}
{age: 16, status: “C”}
{age: 19, status: “O”}
{age: 22, status: “L”}

users collection

{age: 21, status: “U”}
{age: 23, status: “S”}
{age: 25, status: “R”}
{age: 19, status: “O”}
{a

{age: 21, status: “**A**”}
{age: 23, status: “**A**”}
{age: 25, status: “**A**”}
{age: 19, status: “O”}
”}

{age: 21, status: “**A**”}
{age: 23, status: “**A**”}
{age: 25, status: “**A**”}
{age: 19, status: “**A**”}
{age: 22, status: “**A**”}

Write finishes

r

```
  { age: { $gt: 20 } }
)
```

{age: 21, status: “**A**”}
{age: 23, status: “**A**”}
{age: 25, status: “**A**”}
{age: 22, status: “L”}

Read returned documents

Write Operation – Isolation Scenario

A write query comes

```
db.users.updateMany(
  { age: { $gt: 18 } },
  { $set: { status:  “A”, $isolated: 1 } }
)
```

{age: 21, status: “U”}
{age: 23, status: “S”}
{age: 17, status: “E”}
{age: 25, status: “R”}
{age: 15, status: “S”}
{age: 16, status: “C”}
{age: 19, status: “O”}
{age: 22, status: “L”}

users collection

{age: 21, status: “U”}
{age: 23, status: “S”}
{age: 25, status: “R”}
{age: 19, status: “O”}
{a

{age: 21, status: “**A**”}
{age: 23, status: “**A**”}
{age: 25, status: “**A**”}
{age: 19, status: “O”}
”}

{age: 21, status: “**A**”}
{age: 23, status: “**A**”}
{age: 25, status: “**A**”}
{age: 19, status: “**A**”}
{age: 22, status: “**A**”}

Write finishes

```
r
    { age: { $gt: 20 } }
)
```

Read has to wait

Read returns the results

{age: 21, status: “**A**”}
{age: 23, status: “**A**”}
{age: 25, status: “**A**”}
{age: 22, status: “**A**”}

References

- MongoDB online documents:
  - Mongo DB Data Models
    - http://docs.mongodb.org/manual/core/data-modeling-introduction/
  - MongoDB CRUD Operations
    - http://docs.mongodb.org/manual/core/crud-introduction/
  - Pramod J. Sada distilled, Addison-Wesley Professi 012)
    - https://www.amazon.com/NoSQL ging-Polyglot -Persistence/dp/0321826620