COMP5338 – Advanced Data Models

Week 2: Document Store: Data Model and Simple Query

Dr. Ying Zhou
School of Computer Science

Outline

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

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Structured and Unstructured Data

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

Supplier Table:

SuppID	Name	Phone
8703	Heinz	0293514287
8731	Edgell	0378301294
8927	Kraft	0299412020
9031	CSR	0720977632

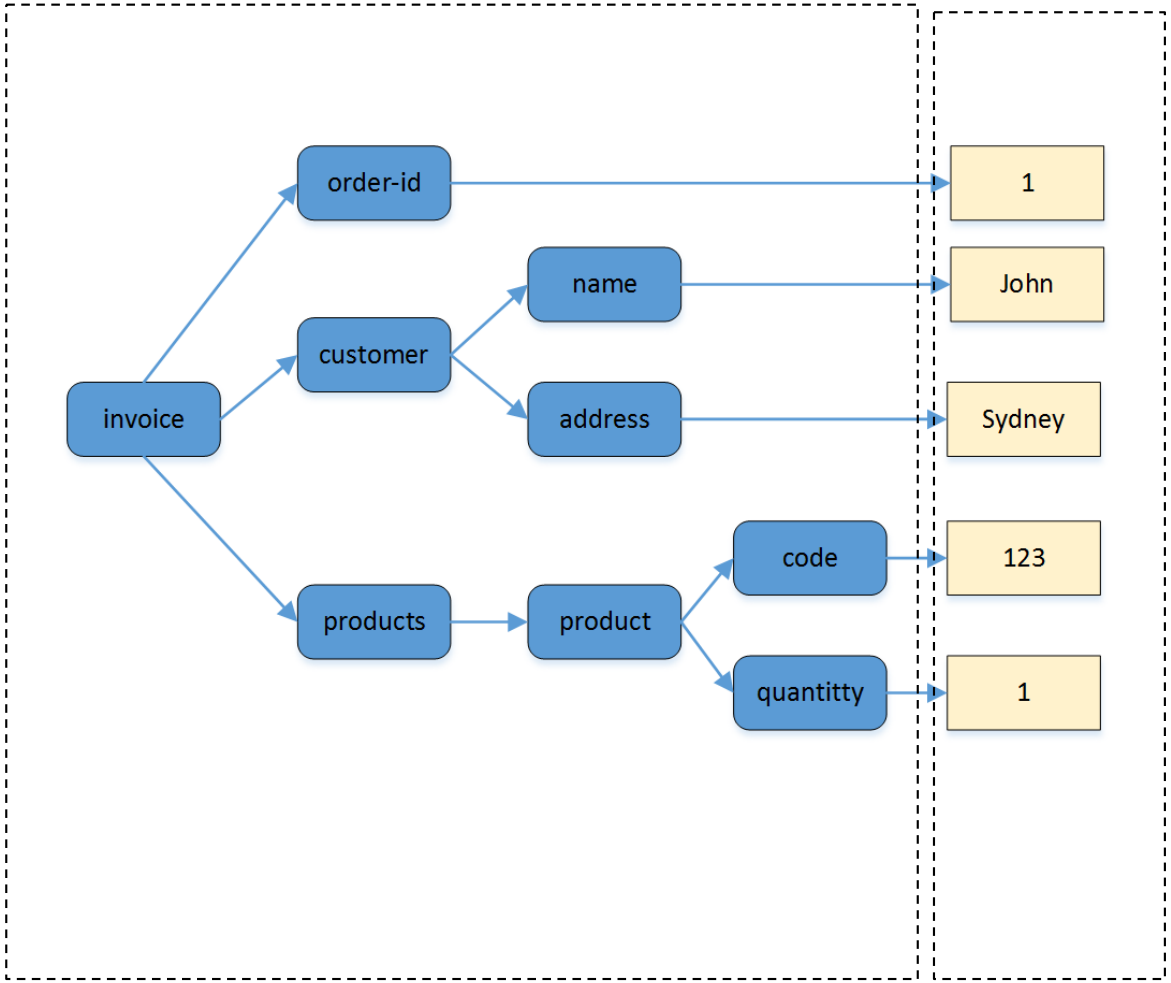
- **Unstructured data** does not follow any predefined "model" or "format" that is *aware to the underlying system* . Examples include data stored in various files, e.g word document

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 classic relational model effectively
- **Self-describing** capability is the key feature of semi-structured data
  - schema/structure is an integral part of the data, instead of a separate declaration
  - in relational database system, the structure is "**declared**" when a table is created. All rows in the table need to follow the structure.
- **XML** and **JSON** are two types of semi-structured data
  - Both provide a way to incorporate the structure as part of the 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>
```

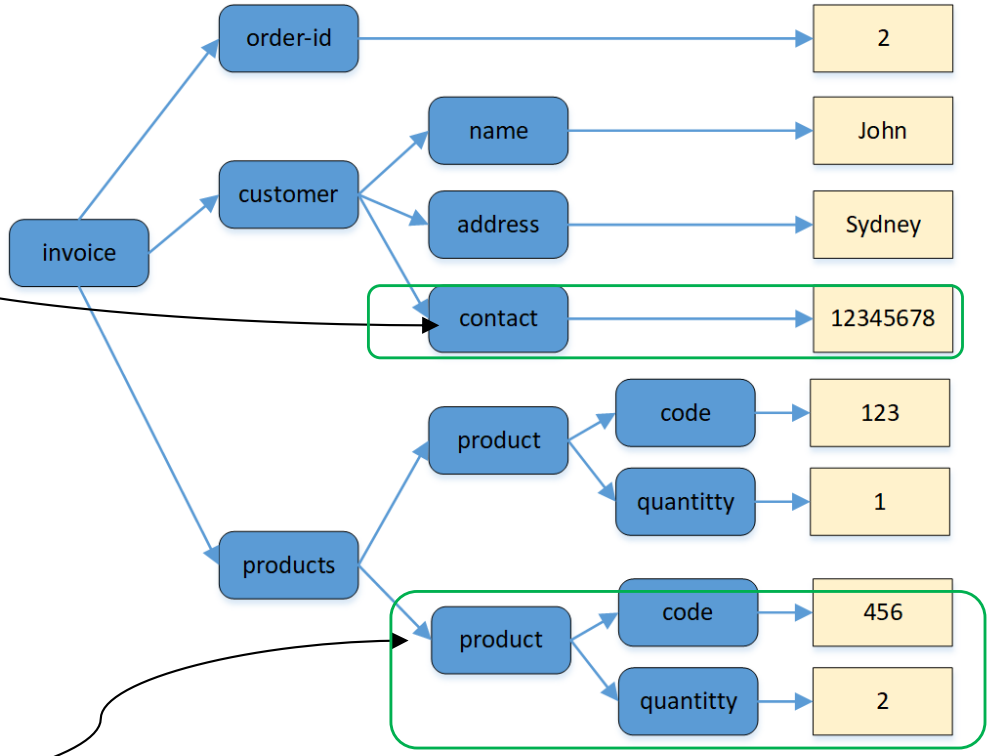


metadata/structure information

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 object is represented as a list of property names and values contained in curly braces, in the following format:

```
{ propertyName1 : value1, propertyName2 : 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}]
            }
```

array

```
Invoice _3= {
            order_id: 3,
            customer: {name: "Smith",
                       address: "Melbourne",
                       contact: "12345"},
            products: [{ code: "123", quantity: 20},
                       { code: "456", quantity:2}]
            delivery: "express"
}
```

Document Store

- Document store or document oriented database stores data in semi-structured documents
  - Document structure is *flexible*
- Provide own query syntax (different to standard SQL)
- Usually has powerful index support
- Examples:
  - XML based database
  - JSON based database: MongoDB

Outline

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

Matching Terms between SQL and MongoDB

MongoDB is a general purpose document store.

SQL	MongoDB
Database	Database
Table	Collection
Row	BSON document
Column	BSON field
Primary key	`_id` field

https://www.mongodb.com/json-and-bson

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

Defined ***once*** during table creation

two rows

Field name is part of ***data***

Repeated in every document

```
{ _id: 12345,
  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,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27
}
```

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: 27,
  address: { number: 1,
             name: “cleveland street”,
             suburb: “chippendale”,
             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 implement them in language-specific way
- 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/

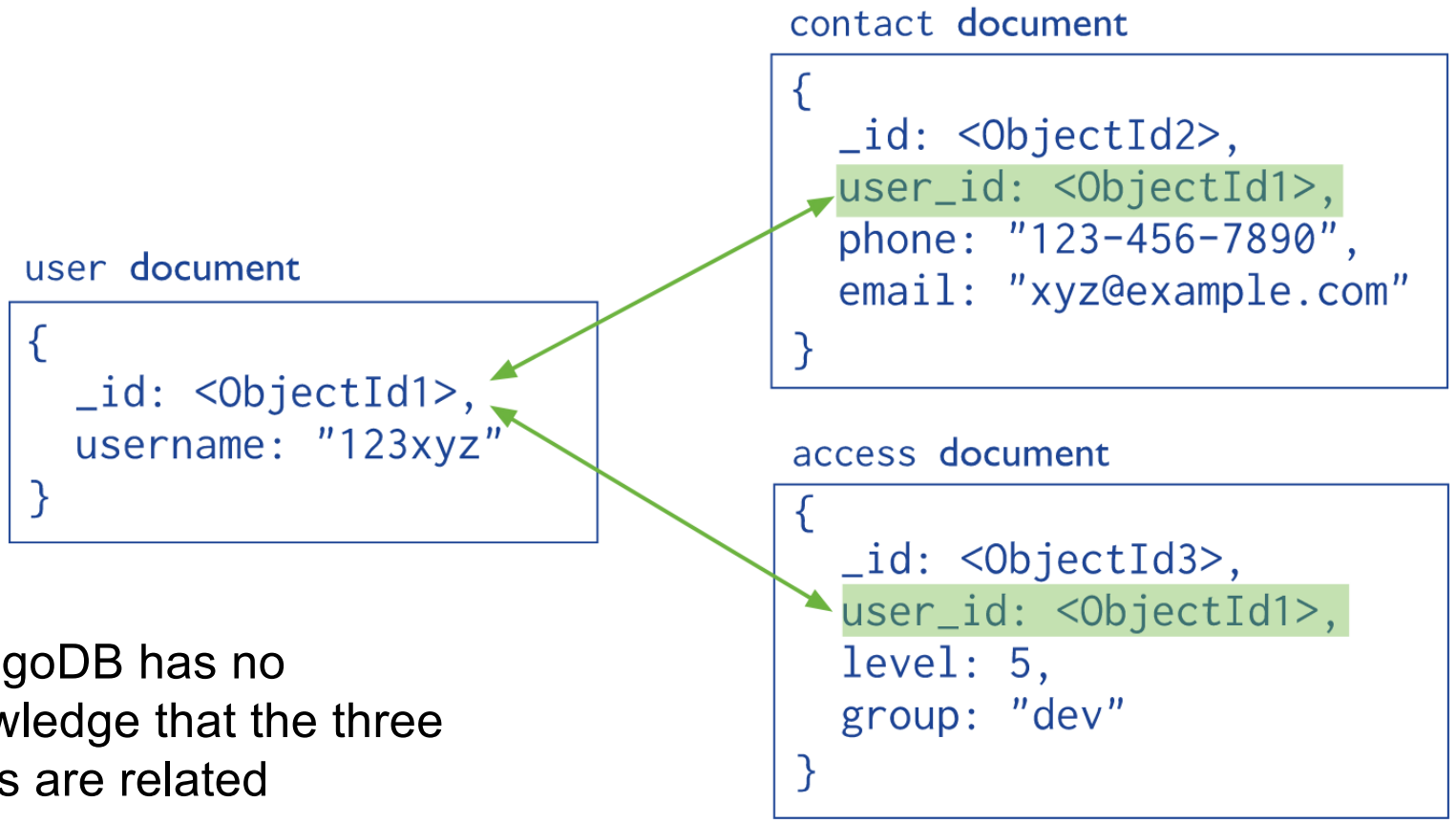
Data Modelling

- Key design decision in MongoDB data modelling involves how to represent *relationship* between data
  - How many collections should we use
  - What is the rough document structure in each collection
- Embedding or Referencing
  - Which object should have its own Collection
    - And reference the **`_id`** in other collection
  - Which object can or should be embedded in other object
- As the database system does not keep schema information, the relationship is "*remembered*" and "*managed*" externally by developers

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.



MongoDB has no knowledge that the three fields are related

Embedding

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

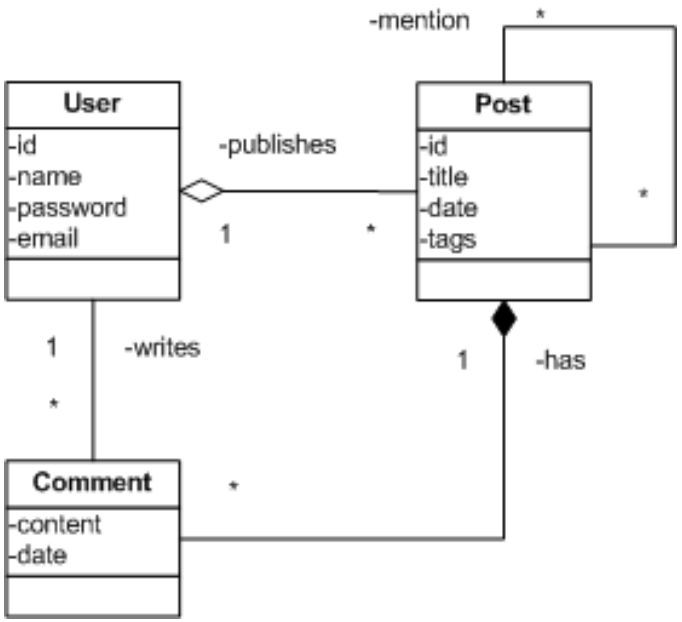
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

`_id` is not required for contact and access document now

“Schema” Design Example



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

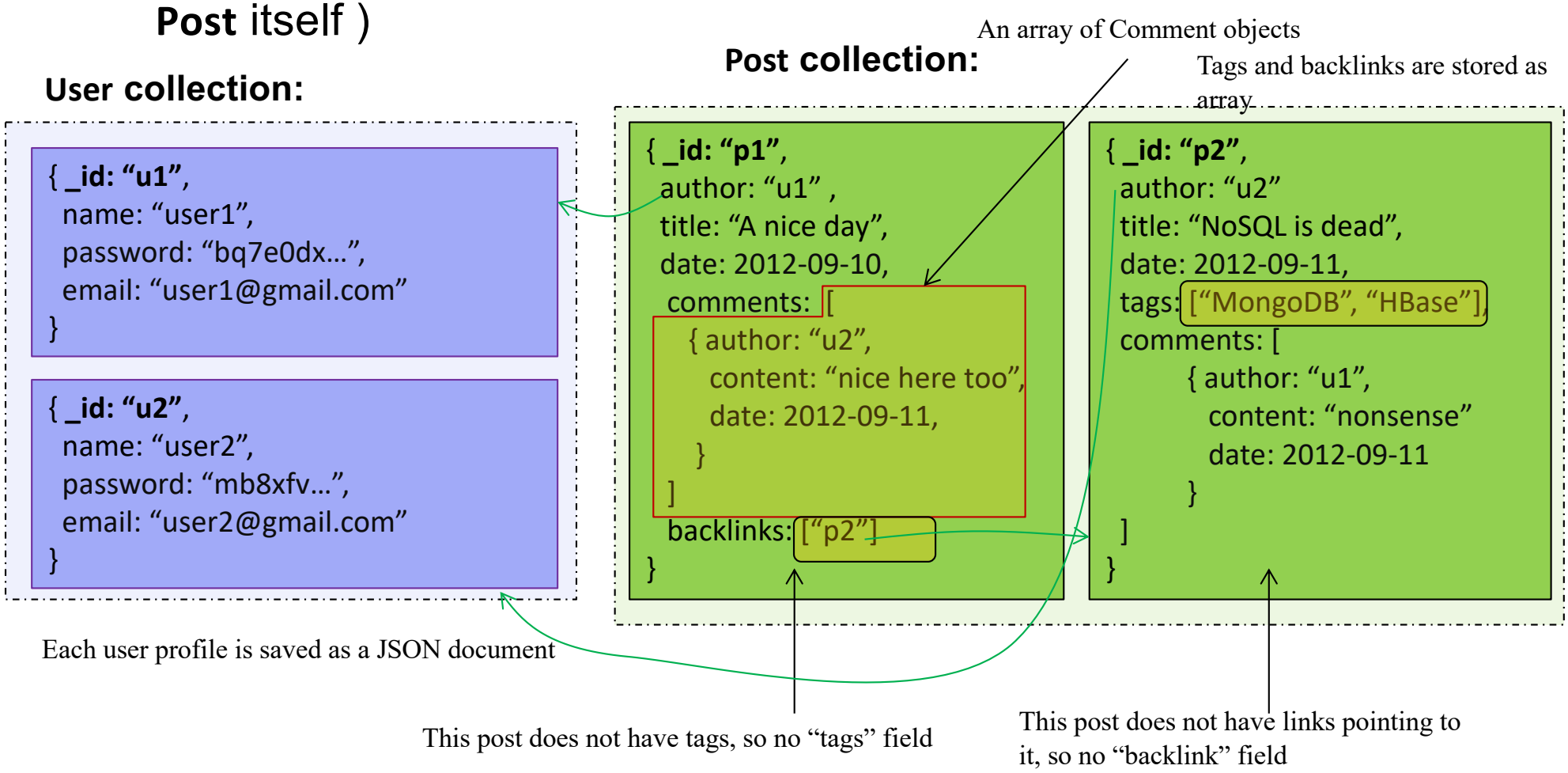
http://docs.mongodb.org/manual/applications/data-models/

MongoDB schema design

- Using **three** collections
  - **User** collection
  - **Post** collection (with links to **User** and **Post** itself)
  - **Comment** Collection(with links to **User** and **Post**)
- Using **two** collections
  - **User** collection
  - **Post** collection (with embedded **Comment** object and links to **User** and **Post** itself

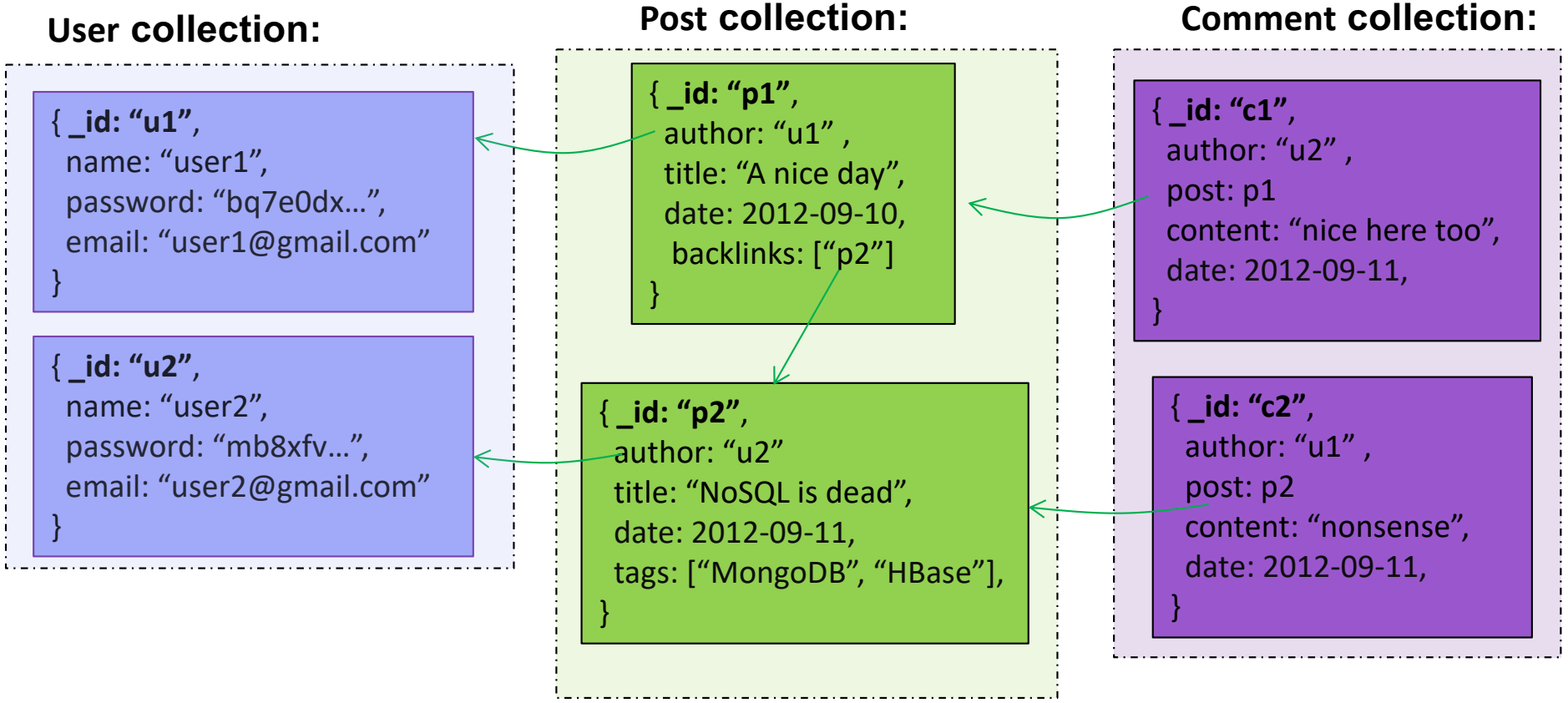
Two Collections Schema

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



Three Collections Schema

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



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 to show all comments when showing a post, or only the latest few, or not at all?
      - Are most comments made in a short period of time?
    - Atomicity consideration
      - Is there "all or nothing" update requirement with respect to post and comment
- Other design variation?
  - In three collection schema, store post-comment link information in **Post** collection instead of **Comment** 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
    - BUT, other features need to be considered
      - ***Post*** and ***Comment*** have a composition relationship, but it might be beneficial to model them as separate documents
  - ***aggregation*** relationship are generally modelled as links (references) with the link data modelled in the 'part' object.
    - Eg. ***Department*** and ***Employee***
  - **Many-to-Many** relationship are generally modelled as links (references)
    - Eg. Course and Students enrolled in a course

Outline

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

MongoDB Queries

- In MongoDB, a ***read*** query targets a specific collection. 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 may *create*, *update* or *delete* data. One query modifies the data of a single collection. Update and delete query can specify query **criteria**

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

Read Operation Interface

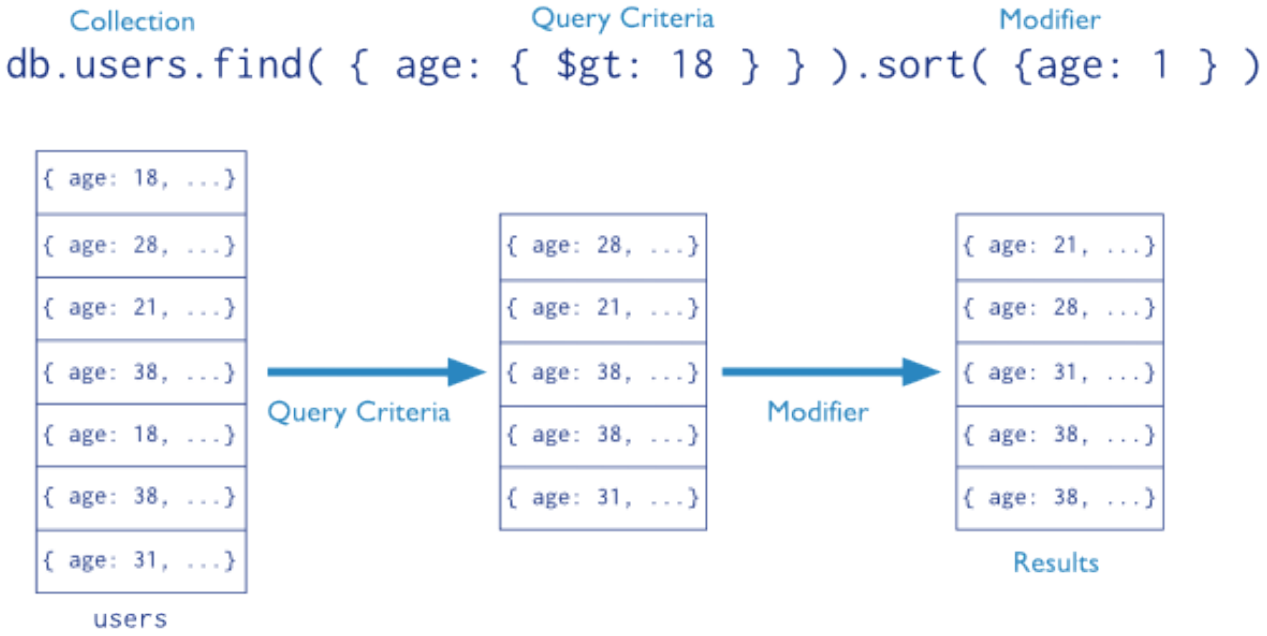
- db.collection.find()

```
db.users.find(                          ← collection
   { age: { $gt: 18 } },                ← query criteria
   { name: 1, address: 1 }              ← projection
).limit(5)                              ← cursor modifier
```

Find at most 5 documents in the **users** collection with **age** field greater than 18, return only the name and address field of each document.

```
SELECT _id, name, address   ← projection
FROM   users                ← table
WHERE  age > 18             ← select criteria
LIMIT  5                    ← 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/JSON document (query object)
  - Individual condition is expressed using predefined selection operator, eg. **$gt** is the operator for "greater than"
- Query **projection** are expressed as BSON/JSON document as well

SQL	MongoDB Query in Shell
select * from user	db.user.find() or db.user.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({“emails.0”: “joe@gmail.com”})
    - Find user(s) whose first email is “joe@gmail.com”.
  - 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

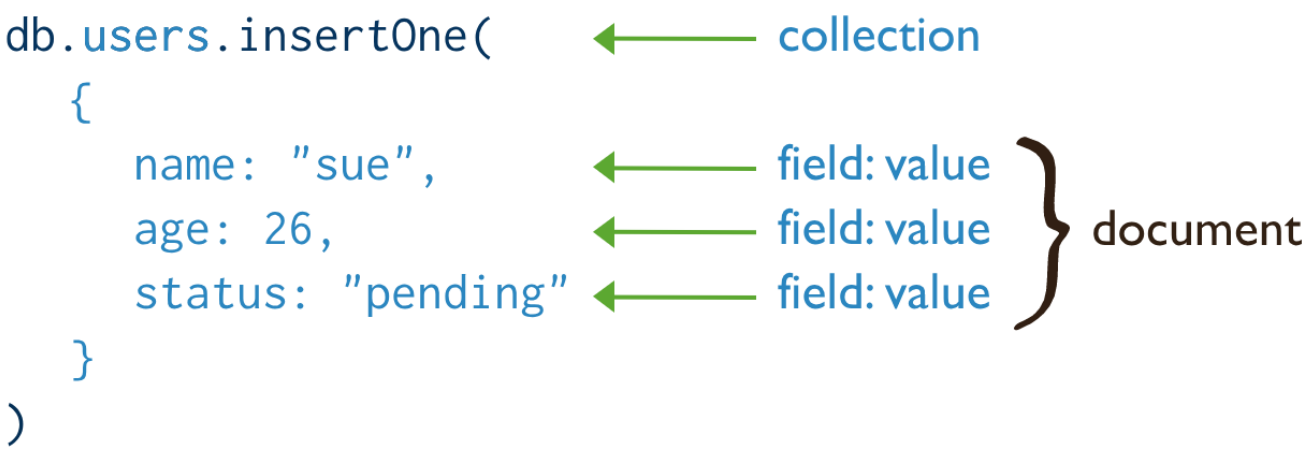
- Embedded Document can be queried as a **whole**, or by **individual field**, or by **combination of individual fields**
  - db.user.find({**address**: {number: 1, name: “pine street”, suburb: “chippendale”, zip: 2008}})
  - db.user.find({**“address.suburb”:** “chippendale”})
  - db.user.find({**“address.name”**: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

user collection

```
{ _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,
  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 document does contain an "**_id**" field, it should have a unique value
- Two other operations:
  - **insertMany**
    - Insert many documents
  - **Insert**
    - Major language APIs only support **insertOne** and **insertMany**

Write Query – Update Operation

```
db.users.updateMany(                          <-- collection
   { age: { $lt: 18 } },                      <-- update filter
   { $set: { status: "reject" } }             <-- update action
)
```

Has the same effect as the following SQL:

```
UPDATE users                    <-- table
SET    status = "reject"        <-- update action
WHERE  age < 18                 <-- update criteria
```

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},
    {$push: {emails: {$each: ["mary@gmail.com", "mary@microsoft.com"]}}})
  - db.user.updateOne({_id: 12345}, {$pull: {emails: "joe@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}

https://docs.mongodb.com/manual/reference/operator/update/push/#up._S_push

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 document matching a certain condition

Atomicity of write operation (single document)

- 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

https://docs.mongodb.com/manual/core/read-isolation-consistency-recency/

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.updateOne(
  { item: "paper" },
  { $set: { "size.uom": "cm",   status: "P" }
  }
)
```

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

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

```
{ 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" }]);
```

Atomicity of write operation (multi documents)

- 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 mechanisms were used in earlier versions
  - The **$isolated** operator can prevents a write operation that affects multiple documents from yielding to other reads or writes once the first document is written
- All those mechanisms have great performance 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"}
{age: 22, status: "L"}

{age: 21, status: "**A**"}
{age: 23, status: "**A**"}
{age: 25, status: "**A**"}
{age: 19, status: "O"}
{age: 22, status: "L"}

write is on going, a read query comes

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

Write finishes

```
db.users.find(
  { age: { $gt: 20 } }
)
```

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

Read returned documents

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. Sadalage, Martin Fowler NoSQL distilled, Addison-Wesley Professional; 1 edition (August 18, 2012)
    - https://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620