VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

MongoDB Data Modeling

Lecturer : Dr. Pavle Mogin

SWEN 432
Advanced Database Design and Implementation

Plan for MongoDB Modelling

- Prologue
- Data Modelling
  - Embedding
  - Referencing
- Data Use and Performance:
  - Indexing
  - ***Reedings:***
    - *Have a look at Readings on the Home Page*

Prologue

- MongoDB is an open source project mainly driven by the company 10gen Inc
- The main goal of the development of MogoDb was to close the gap between fast and highly scalable key-value stores and feature rich RDBMSs
- Features:
  - Document Data Model and a relatively rich query language,
  - Data partitioning by sharding,
  - Master – slave mode of replication,
  - **No data versioning**
- Prominent users:
  - SourceForge.net,
  - Foursquare,
  - New York Times

Data Model

- A MongoDB installation hosts a number of databases
- A ***database*** is a physical data container of a set of collections
- A ***collection*** contains a set of documents
- A ***document*** contains a set of key-value (also referred to as field_name-field_value) pairs
  - The basic unit of data
  - Logically analogous to a JSON object
- Documents have a ***dynamic schema***:
  - There is no predefined schema of a collection,
  - Documents are self describing,
  - In a collection, documents do not need to have the same structure,
  - Common fields belonging to different documents may have different data types

Document

- MongoDB stores all data in documents that are JSON-style data structures composed of key-value pairs:
  - All database records,
  - Query selectors (what records to select for **rud** operations),
  - Update definitions (which fields to modify),
  - Index specifications (what fields to index),
  - Data output by MongoDB
- Documents are stored on disk in the BSON format - a binary representation of JSON with an additional type information

Document Structure

- MongoDB documents are composed of field-and-value pairs and have the following structure:

{

$field_1$: $value_1$, $field_2$: $value_2$, ..., $field_n$: $value_n$

}

- The value of a field can be any of the BSON *data types*, including other documents, arrays, and arrays of documents

A Document Example

```
var class = {
  _id: ObjectId (“509980df3”),
  course: {code: “SWEN432”,
            title: “Advanced DB”},
  year: 2017,
  students: [“Matt”, “Jack”,..., “Lingshu”],
  no_of_st: 27
}
```

- The document contains values of varying types
  - The primary key is `_id` and it is of the `ObjectId` type,
  - The field `course` is a subdocument, and
  - `students` is an array of strings

Field Names

- Field names are strings
- Restrictions on field names:
  - The field names **cannot** start with the dollar sign ($) character,
  - The field names **cannot** contain the dot (.) character,
  - The field names **cannot** contain the null character

Field `_id`

- The field **`_id`** is reserved for use as a document’s primary key
  - Its value must be unique in the collection,
  - It is immutable,
  - May be of any type other than an array or a regular expression type
  - MongoDB creates a unique index on the `_id` field during the creation of a collection.
  - It is always the first field in the documents
- The following are common options for storing values for `_id`:
  - Use an *ObjectId,*
  - Use a natural unique identifier, if available
    - This saves space and avoids an additional index,
  - Generate an auto-incrementing number

BSON Data Types

- MongoDB supports a decent number of built in BSON data types, some of them are:
  - Double
  - String
  - Array
  - Binary data
  - ObjectId
  - Boolean
  - Date
  - Null
  - Regular Expression
  - JavaScript
  - Integer (32-bit and 64-bit)

ObjectId

- *ObjectId* is a 12-byte *BSON* type, constructed using:
  - A 4-byte value representing the seconds since the Unix epoch,
  - A 3-byte machine identifier,
  - A 2-byte process id, and
  - A 3-byte counter, starting with a random value
- ObjectIds are small, most likely unique, and fast to generate
- MongoDB uses ObjectIds as the default value for the `_id` field if the `_id` field is not specified by a client
- Additional benefits of using ObjectIds for the `_id` field:
  - In the mongo shell, you can access the creation time of the ObjectId, using the `getTimestamp()` method,
  - Sorting on ObjectId values is roughly equivalent to sorting by creation time.

Constructing Values in mongo shell

- To generate a new `ObjectId`, use the `ObjectId()` constructor with no argument:

```
var x = ObjectId()
```

- In this example, the value of `x` would be:

```
ObjectId("507f1f77bcf86cd799439011")
```

- To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` :

```
ObjectId("507f191e810c19729de860ea").getTimestamp()
```

- This operation will return the following `Date` object:

```
ISODate("2012-10-17T20:46:22Z")
```

- Construct a date using the **`new Date()`** constructor:

```
var mydate = new Date()
```

- Find the month portion of the `mydate` value

```
mydate.getMonth()
```

The Key Data Modelling Decisions

- The key challenge in data modeling is balancing:
  - The needs of the application (queries, updates, data processing),
  - The performance characteristics of the database engine, and
  - The data retrieval patterns
- The key decision in designing a data model for MongoDB is how to represent relationships between data objects
- Two representation mechanisms:
  - Embedding and
  - Referencing

Embedded Relationships (Example)

```
{
_id: “SWEN432”
 title: “Advanced Databases”,
 coordinator:     {
                   name:   “Pavle”,
                   email: “pmogin@ecs.vuw.ac.nz”
                  },


 guest_lecturer: {
                   name:   “Aaron”
                   email: “aaron@thelastpickle.com
                  },

 year:       2014,
 trimester: 1
}
```

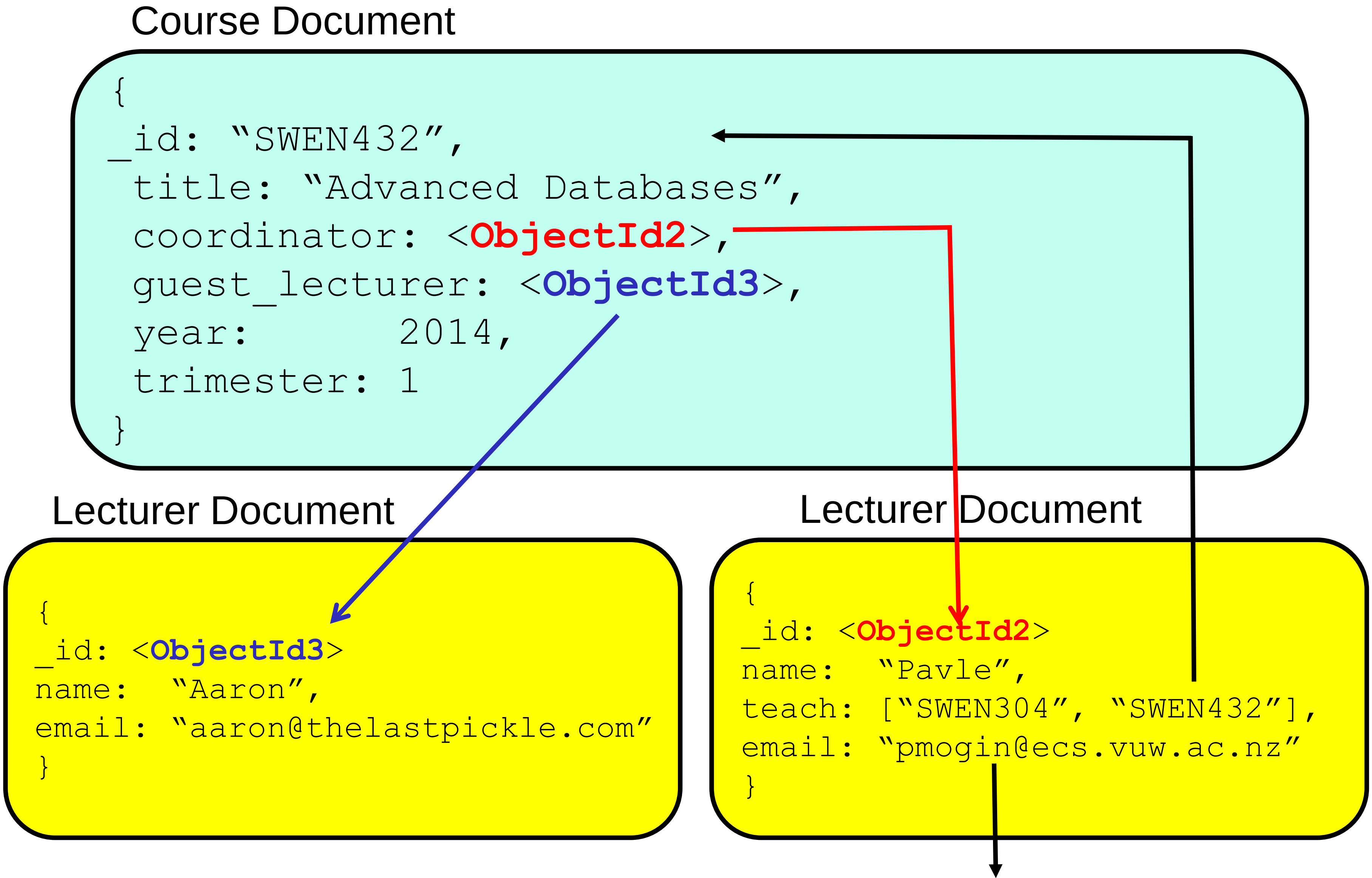
Embedded Data

- Embedded documents capture relationships by storing related data objects within a single document
  - Subdocuments may be stored in fields or an array within another document
  - Leads to ***denormalization***
- Denormalized data structures allow reading and manipulating related data in a single db operation
- Embedding is the preferred technique in the case of:
  - *One – to – one* relationships, and
  - *One – to – many* relationships with no extensive overlapping of objects on the *many* side
- A potential disadvantage of embedding is a possibly uncontrolled growth of a document through adding new objects on the *many* side

Referencing

- Assume each lecturer teaches a number of courses
  - Then embedding may lead to a considerable data redundancy
- An alternate approach is to store course and lecturer objects as separate documents and link them using references (document keys)
- In principle, references can be stored in the object on the *one* side, or in objects on the *many* side (or even on both sides)
  - Query patterns and the growth of the number of links influence the decision of link placement

Relationship by Referencing (Example)



Course Document

```
{
_id: "SWEN432",
 title: "Advanced Databases",
 coordinator: <ObjectId2>,
 guest_lecturer: <ObjectId3>,
 year:       2014,
 trimester: 1
}
```

Lecturer Document

```
{
_id: <ObjectId3>
name:  "Aaron",
email: "aaron@thelastpickle.com"
}
```

Lecturer Document

```
{
_id: <ObjectId2>
name:  "Pavle",
teach: ["SWEN304", "SWEN432"],
email: "pmogin@ecs.vuw.ac.nz"
}
```

Implementing Referencing

- MongoDB applications use one of two methods for relating documents:
  - ***Manual references*** where you save the `_id` field of one document in another document as a reference
    - These references are simple and sufficient for most use cases
  - The other method is to use ***DBRefs***
- MongoDB documentation recommends using manual references

Using Manual References

```
use mydb
var coordinator_id = ObjectId()
var guest_lec_id = ObjectId()

db.class_ref.insert({
_id: “SWEN432”,
 title: “Advanced Databases”,
 coordinator: coordinator_id,
 guest_lecturer: guest_lec_id,
 year:      2014,
 trimester: 1
})
```

When to Use Referencing

- Representing relationships by referencing produces ***normalized*** data models
- Referencing is a preferred technique:
  - When embedding leads to data redundancy but does not provide sufficient read performance advantages to outweigh the consequences of data duplication,
  - For representing many – to many relationships, and
  - To represent large hierarchical structures
- Referencing provides a more flexible data model than embedding at the expense of issuing follow-up queries to resolve references

Data Use and Performance

- The following phenomena and mechanisms influence performance of operations on MongoDB databases:
  - Atomicity of writes,
  - Document Growth,
  - Sharding,
  - Indexes, and
  - Capped Collections
- We comment here all of them except sharding
  - Sharding is considered within MongoDB Architecture

Atomicity of Writes

- In MongoDB, write operations are atomic at the document level
  - A single write operation affects just one document within a single collection
- An embedded data model combines all related data for an entity in a single document
  - A single write operation inserts or updates all entity data
- A normalized data model splits data across several documents (and possibly collections)
  - Inserting a single entity requires several write operations that are not atomic collectively
- However, embedding results in less flexible schemes
  - A single entry point to data,
  - Hard to modify applications

Document Growth

- Some updates as:
  - Pushing elements to an array, or
  - Adding new fields

increase a document's size

- If the document's size exceeds the allocated space, MongoDB relocates the document on disk
- Relocation takes longer than in place updating and may lead to space fragmentation
- To avoid relocation, referencing instead of embedding should be used

Indexes

- MongoDB automatically creates a unique index on the `_id` field
- Indexes on fields (other than `_id`) that appear often in queries improve performance for common queries
- Indexes are built as BTrees (facilitating range queries)
- Adding an index has some negative performance impact for write operations
  - For collections with high write-to-read ratio, indexes are expensive since each insert must also update any indexes
- Collections with high read-to-write ratio often benefit from additional indexes

createIndex() or ensureIndex()

- `db.collection.createIndex(keys, options)`
- Parameters:
  - `keys` of the type document:
    - For each field to index, a key-value pair with the field and the index order: `1` for ascending or `-1` for descending
  - `options` of the type `document` (optional)
- The most important options:
  - `unique` of the type `Boolean`:
    - The default value is false
  - `name` of the type `string`:
    - If unspecified, MongoDB generates an index name

```
db.collection.ensureIndex(
{_id: 1, year: -1}, {unique: true}
)
```

Capped Collection (*)

- *Capped collections* are fixed-size collections that support those high-throughput operations that insert and retrieve documents based on insertion order
- Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection

Summary

- MongoDB data model: document oriented
  - A database is a container for a number of document collections,
  - Documents of a collection may have but don’t have to have the same structure,
  - A document is a set of key-value pairs in JSON format,
  - There are no field constraints and no referential integrity constraints,
  - The unique constraint is supported via `createIndex()` method
- The main issue with data modeling is how to represent relationships between entities
  - Embedded relationships
  - Relationships by references