Department of Computer Science

PURDUE UNIVERSITY

MongoDB

Elisa Bertino and Spencer Pearson

Presentation based on
•**slides "Document Databases"** by *David Novak, FI, Masaryk University, Brno*
http://disa.fi.muni.cz/david-novak/teaching/nosql-databases-2017/
•**Slides "MongoDB"** by Pietro Colombo, University of Insubria

Department of Computer Science

PURDUE UNIVERSITY

What Is It?

- A document-oriented database
  - documents encapsulate and encode data in some standard formats
- NoSQL database
  - non-adherence to the widely used relational database systems
  - highly optimized for retrieve and append operations
- It uses BSON format
- It is schema-less
  - No more configuring database columns with types
  - Documents are self-describing
  - Documents are analogous to structures in programming languages that associate keys with values
  - The values of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents
- No transactions
- No joins

```
{
  name: "sue",                          <-- field: value
  age: 26,                              <-- field: value
  status: "A",                          <-- field: value
  groups: [ "news", "sports" ]          <-- field: value
}
```

PURDUE UNIVERSITY

Why Document Databases

- XML and JSON are popular for data exchange
  - Recently mainly JSON
- Data stored in document DB can be used directly

- Databases often store objects from memory
  - Using RDBMS, we must do Object Relational Mapping (ORM)
    - ORM is relatively demanding
  - JSON is much closer to structure of memory objects
    - It was originally for JavaScript objects
    - Object Document Mapping (ODM) is faster

PURDUE UNIVERSITY

JSON and BSON

JSON

- Text-based open standard for data interchange
  Serializing and transmitting structured data
- JSON = JavaScript Object Notation
  Derived from JavaScript scripting language
  Uses conventions of the C-family of languages
- Filename: *.json
- Language independent

(see www.json.org)

BSON

It is binary-encoded serialization of JSON documents

JSON representation of record describing a person

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

Department of Compu

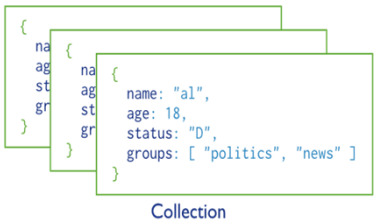
PURDUE UNIVERSITY

Terminology

mongoDB

RDBMS	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id

- each JSON document:
  - belongs to a collection
  - has a field _id
    - unique within the collection
- each collection:
  - belongs to a "database"



Collection

http://www.mongodb.org/

Department of Compu

PURDUE UNIVERSITY

Document

mongoDB

- A document is a set of keys (also called field names) with associated values (no duplicate keys!!)
- Represented as JSON objects: *{"greeting" : "Hello, world!"}*
- Serialized as a BSON object
- Documents contain multiple key/value pairs: *{"greeting" : "Hello, world!", "foo" : 3}*
- Restrictions on keys:
  - ❑ The key cannot start with the $ character
    - Reserved for operators
  - ❑ The key cannot contain the . character
    - Reserved for accessing sub-fields
- Every document must have field _id
  - ❑ Used as a primary key
  - ❑ Unique within the collection
  - ❑ Immutable
  - ❑ Any type other than an array
  - ❑ Can be generated automatically
- The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents

NOTE: the double quotes around the key can be omitted

Data Types

- *null:* can be used to represent both a null value and a nonexistent field:

  *{"x" : **null**}*
- *Boolean:* fields can be set to *true* and *false*:

  *{"x" : **true**}*
- *number:*

  If not specified the shell uses 64-bit floating point numbers

  {"x" : 3.14} or {"x" : 3}

  4-byte or 8-byte signed integers require the classes *NumberInt* or *NumberLong*

  {"x" : NumberInt("3")} or *{"x" : NumberLong("3")}*
- *string:* any sequence of UTF-8 characters

  {"x" : "foobar"}

Department of Computer Science
PURDUE UNIVERSITY

Data Types

- *date:* stored as milliseconds :

  *{"x" : **new** Date()}*
- *regular expression:* using JavaScript's regular expression syntax:

  {"x" : /foobar/i}
- *Array:* lists of values can be represented as arrays:

  {"x" : ["a", "b", "c"]}
- *embedded document:* Documents can contain entire documents embedded as values in a field

  {"x" : {"foo" : "bar"}}

PURDUE UNIVERSITY

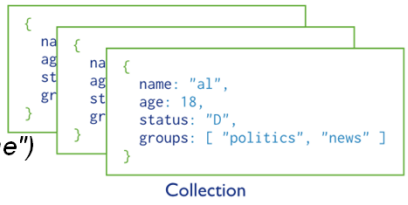
Data Types

- *binary data:* strings of arbitrary bytes
- *code:* queries and documents can include arbitrary JavaScript code: {"x" : *function*() { /* ... */ }}
- *object id:* 12-byte ID for documents.
  {"x" : ObjectId()} this method creates and returns an ObjectID
- An example: **507f1f77bcf86cd799439011**
- MongoDB documents must have an *"_id"* key.
- *"_id"* can be of any type, but it defaults to an *ObjectId*.
- In a collection, every document must have a unique value for *"_id"*

PURDUE UNIVERSITY

Collections

- A *collection* is a group of related documents
- Collections can be thought of as the analog to a tables in relational databases
- Collections have *dynamic schemas:*
  The documents within a single collection can have different structures
  {"greeting" : "Hello, world!"}
  {"foo" : 5}
- Collections are created using
  - the method *db.createCollection("name")*
    db.createCollection("myCollection")
  - implicitly by using it; if a collection that does not exist is referenced in a command, the system automatically creates it
    db.myCollection2.insert({{"name" : "Max"})



PURDUE UNIVERSITY

Databases

- MongoDB groups collections into *databases*
- A single instance of MongoDB can host several databases
- Each database groups together zero or more collections
- A database has its own permissions, users and roles and each database is stored in separate files on disk
- A good rule is storing all data for a single application in the same database
- Reserved database names:
  - *admin:* the "root" database, in terms of authentication. If a user is added to the *admin* database, the user automatically inherits permissions for all databases.
  - *Config:* used for storing information of sharded setup

PURDUE UNIVERSITY

Database Schema

mongoDB

- Documents have flexible schema
  - Collections do not enforce specific data structure
- Key decision of data modeling:
  - References vs. embedded documents
  - In other words: Where to draw lines between aggregates
    - Structure of data
    - Relationships between data
- Embedded Documents
  - Related data in a single document structure
    - Documents can have subdocuments (in a field or array)

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

Embedded sub-document

Embedded sub-document

Embedded Docs

mongoDB

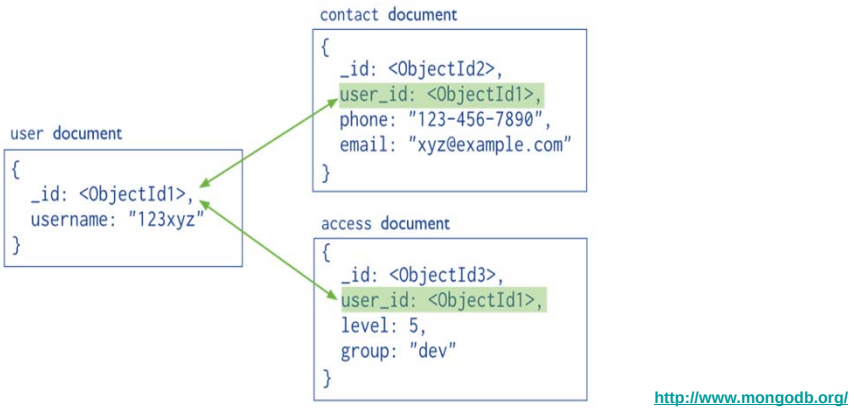
- Denormalized schema
- Main advantage:
  Manipulate related data in a single operation
- Use this schema when:
  - One-to-one relationships: one doc "contains" the other
  - One-to-many: if children docs have one parent document
- Disadvantages:
  - Documents may grow significantly during the time
  - Impacts both read/write performance
    - Document must be relocated on disk if its size exceeds allocated space
    - May lead to data fragmentation on the disk

Department of Compu
PURDUE UNIVERSITY

References

mongoDB

- Links/references from one document to another
- Normalization of the schema



http://www.mongodb.org/

References

mongoDB

- More flexibility than embedding
- Use references:
  - When embedding would result in duplication of data
    - and only insignificant boost of read performance
  - To represent more complex many-to-many relationships
  - To model large hierarchical data sets
- Disadvantages:
  - Can require more roundtrips to the server
    - Documents are accessed one by one

Write Operations

- Write operations create or modify data in the MongoDB instance
  - target a single *collection*
  - atomic on the level of a single *document*
- There are three types of write operations: *insert, update and remove*.
- Write operations cannot affect more than one document atomically
- Update and remove operations allow one to specify selection criteria for identifying the documents to update or remove.
  - Selection criteria are the same as read operations

```
db.users.insert (          <-- collection
   {
     name: "sue",          <-- field: value
     age: 26,              <-- field: value
     status: "A"           <-- field: value
   }                       } document
)
```

Equivalent to

```
INSERT INTO users                <-- table
            ( name, age, status ) <-- columns
VALUES      ( "sue", 26, "A" )    <-- values/row
```

Insert

- Consider the document

{"title" : "My Blog Post",
"content" : "Here's my blog post.",
"date" : new Date()}

- We assign it to variable post

>post={"title" : "My Blog Post",
"content" : "Here's my blog post.",
"date" : new Date()}

- We use *insert* to store *post* into collection *blog*

>db.blog.insert(post)

- Before storing the document an "_id" key is added to the document (if one does not already exist)

Department of Computer Science
PURDUE UNIVERSITY

Insert

- If a new document *d* is added without the *_id* field, MongoDB
  - adds an *_id* field
  - initializes the field with a unique ObjectId
- The value of the _id field must be unique within the collection
- Any attempt to create a document with a duplicate *_id* value results into a duplicate key exception

PURDUE UNIVERSITY

Read

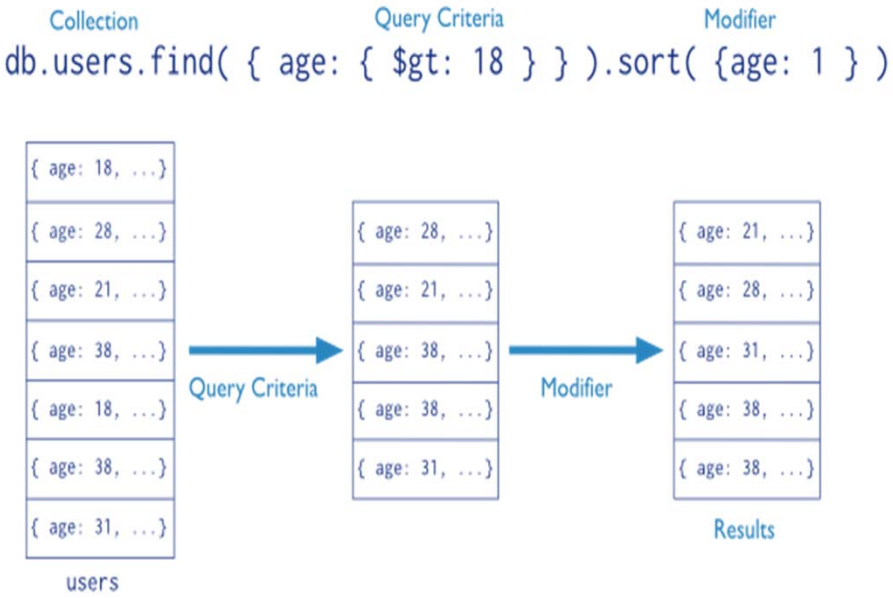
- The post has been saved to the database. We can see it by calling *find* on the collection *blog*:

```
> db.blog.find()
{
"_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
"title" : "My Blog Post",
"content" : "Here's my blog post.",
"date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

- The method find is the method to be used for querying the database
- Definition: db.collection.find(**query**, **projection**)
- When used without input (or with input an empty set {}), it returns the entire collection

PURDUE UNIVERSITY

Query with Predicates



PURDUE UNIVERSITY

Projection

- The second argument passed to *find* specifies the keys to be projected/pruned out from the selected documents
- In order to be projected/pruned out a key the corresponding document key parameter must be set to 1/0

> db.users.find({}, {"username" : 1, "email" : 1})

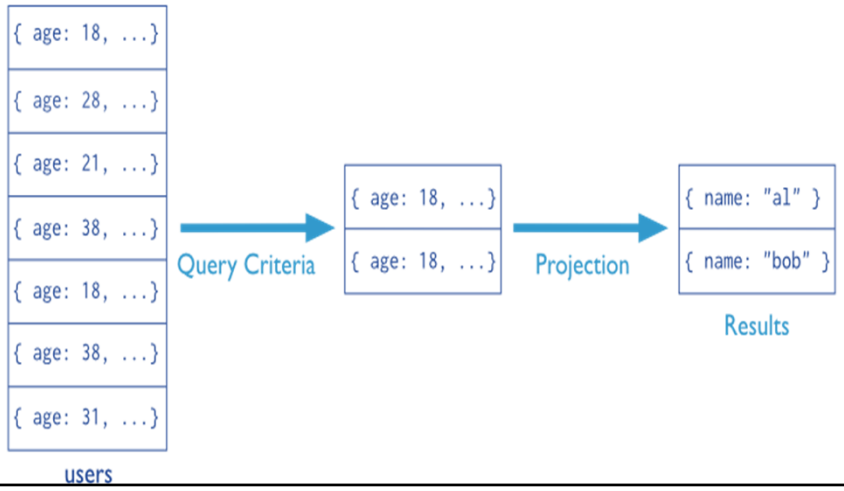
{"_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
"username" : "joe",
"email" : "joe@example.com"}

PURDUE UNIVERSITY

Query with Predicates and Projection

Collection Query Criteria Projection

```
db.users.find( { age: 18 }, { name: 1, _id: 0 } )
```



PURDUE UNIVERSITY

Projection

- By default, the *_id* field is included in the results.
  - To suppress the *_id* field from the result set, specify *_id: 0* in the projection document
- Exclude One Field From a Result Set

➢ *db.records.find({ "user_id": { $lt: 42 } }, { "history": 0 })*

- Return Two fields and the _id Field

➢ *db.records.find({ "user_id": { $lt: 42 } }, { "name": 1, "email": 1 })*

- Return Two Fields and Exclude _id

> db.records.find({ "user_id": { $lt: 42} }, { "_id": 0, "name": 1 , "email": 1 })

PURDUE UNIVERSITY

Selection criteria

- Selection of documents with specific keys
- Querying for a simple type is as easy as specifying the value that you are looking for.
  e.g., find all documents where the value for "age" is 27
  > db.users.find({"age" : 27})
- Multiple conditions are specified by adding more key/value pairs to the query document, which gets interpreted as "*condition1* AND *condition2* AND … AND *conditionN*."

  e.g, find all users who are 27-year-olds with the username "joe,"
  > db.users.find({"username" : "joe", "age" : 27})

PURDUE UNIVERSITY

Example

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

Equivalent to the SQL query

```
SELECT _id, name, address  ← projection
FROM   users               ← table
WHERE  age > 18            ← select criteria
LIMIT  5                   ← cursor modifier
```

Department of Computer Science

PURDUE UNIVERSITY

Query Conditions

- Query Conditions: make use of *comparison* operators *"$lt", "$lte", "$gt",* and *"$gte"* corresponding to <,<=, >, and >=, respectively
- The operators can be combined to look for a range of values.
  e.g, find users with an age between 18 and 30
  > db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
- Conditions are implicitly conjuncted: *age>=18 and age<=30*
- The operator "$ne“ is used to query for documents where a key’s value is not equal to a certain value
  e.g., find all users who do not have the username *“joe”*
  > db.users.find({"username" : {"$ne" : "joe"}})

PURDUE UNIVERSITY

Query Conditions

- Operator "$in" is used to specify multiple match values for a single key
- E.g., suppose we were running a raffle and the winning ticket numbers were 725, 542, and 390. To find these documents:

  >db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})

- If "$in" is given an array with a single value, it behaves the same as directly matching

  >*db.raffle.find(*{ticket_no : {$in : [725]}})

  matches the same documents as

  >*db.raffle.find(*{ticket_no : 725})

Department of Computer Science

PURDUE UNIVERSITY

Query Conditions

- The operator "$nin" returns documents that don't match any of the criteria in the array.

  e.g., return all of the people who didn't win anything in the raffle

  > db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})
- Different from *$in,* the operator *"$or"* takes an array of possible criteria possibly referring to multiple keys
- e.g. find documents where "ticket_no" is 725 or "winner" is true

  > db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : ***true****}]})*

PURDUE UNIVERSITY

Query Conditions

- "$or" can contain other conditions

  e.g., match any of the three "ticket_no" values or the "winner" key

  *> db.raffle.find({"$or" : [{"ticket_no" : {"$in" : [725, 542,390]}},{"winner" : **true**}]})*

- The operators “$and“ and "$nor“ follow similar criteria

```
db.inventory.find(
{ $and : [
        { $or : [ { price : 0.99 }, { price : 1.99 } ] },
        { $or : [ { sale : true }, { qty : { $lt : 20 } } ] }
        ]
} )
```

Department of Computer Science

PURDUE UNIVERSITY

Query Conditions

- "$not" can be applied on top of other operators.

As an example, let’s consider the modulus operator, "$mod“ which queries for keys whose values, when divided by the first value given, have a remainder of the second value

e.g. find users with "id_num“s of 1, 6, 11 and so on.

> db.users.find({"id_num" : {"$mod" : [5, 1]}})

To return users with "id_num"s of 2, 3, 4, 5, 7, 8, 9, 10, 12, we can use "$not":

> db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})

PURDUE UNIVERSITY

Queries for Null or Missing Values

- *null* does match itself

- Suppose to have a collection with the following documents:

➢ *db.c.find()*
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }

- To find documents whose "y" key is null

> db.c.find({"y":null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y":null }

Department of Computer Science

PURDUE UNIVERSITY

Queries for Null or Missing Values

- Different query operators treat null values differently
- Example:

```
db.inventory.insertMany([ { _id: 1, item: null }, { _id: 2 } ])
/* insertMany creates and insert multiple documents */
```

- db.inventory.find({ item: null })
  - this query retrieves all documents that: (i) contain the key “item” and key has a null value; OR (ii) do not contain the key “item”
  - Both documents 1 and 2 are retrieved
- db.inventory.find({ item : { $type: 10 } })
  - This query finds only the documents that: contain the key “item”; AND (ii) the value of key is null (e.g., the value is of BSON type Null (type number 10)
  - Only document 1 is retrieved
- db.inventory.find({ item : { $exists: false } })
  - This query finds all documents that do not contain the key “item”
  - Only document 2 is retrieved
  - $exists: true would retrieve all documents that contain the key “item” (and thus only document 1 would be retrieved)

Regular Expressions

- Useful for flexible string matching
- E.g. find users with the name Joe or joe, we use a regular expression to do case insensitive matching:
  > db.users.find({"name" : /joe/i})
- In order to match also joey, we can specify a regular expression as
  > db.users.find({"name" : /joey?/i})
- MongoDB uses the Perl Compatible Regular Expression (PCRE); any regular expression syntax allowed by PCRE is allowed

Queries on Arrays

- Designed to behave the way querying for scalars does.
  E.g, if the array is a list of fruits, like this:
  > db.food.insert({"fruit" : ["apple", "banana", "peach"]})
  the query
  >db.food.find({"fruit" : "banana"})
  matches the document
- The document is selected if at least one cell of the array is set to the selected value
- $all is used to match arrays by more than one element
  E.g., suppose we created a collection with three elements:
  > db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
  > db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
  > db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
  In order to find all documents with both "apple" and "banana" :
  > db.food.find({fruit : {$all : ["apple", "banana"]}})
  {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
  {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}

  Order does not matter

PURDUE UNIVERSITY

Queries on Arrays

Exact match also works but it requires no missing or superfluous values in the array

> *db.food.find({"fruit" : ["apple", "banana", "peach"]})*

matches

> *db.food.find({"fruit" : ["apple", "banana"]})*

does not match

- Arrays can be queried by index

> *db.food.find({"fruit.2" : "peach"})*

PURDUE UNIVERSITY

Queries on Embedded Documents

- Two strategies:

  Case 1: querying for the whole embedded document

  Case 2: querying for its individual key/value pairs

- Example:

```
db.inventory.insertMany( [
        { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
        { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
        { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
        { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
        { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

- To specify an equality condition on a field that is an embedded document, one must use the query filter document {<field>: <value>} where <value> is the embedded document to match
- Example of Case 1:

```
db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
/* this query retrieves all documents that have the specified size */
```

Which is the result?

PURDUE UNIVERSITY

Queries on Embedded Documents

- Case 1: Equality matches on the whole embedded document require an *exact* match of the specified <value> document, including the field order.
- For example, the following query does not match any documents in the inventory collection:
  - db.inventory.find({ size: { w: 21, h: 14, uom: "cm" } })
- Case 2: To specify a query condition on fields of embedded documents, the dot notation must be used
- Example:

```
db.inventory.find( { "size.uom": "in" } )
```


PURDUE UNIVERSITY

Queries on References

- MongoDB does not automatically resolve references into documents
- Two options:
  1. Perform additional queries to manually resolve the references
     - Simple, but requires multiple queries
  2. Use $lookup to perform a join
     - Performs the operation in one query, but can be complicated

PURDUE UNIVERSITY

Queries on References

- To perform additional queries, save the results of a query as a variable
  - find().toArray() – stores the result of a find() as an array which can be compared to
  - findOne() – stores the result of a find() operation that returns a single document as an array
- In the previous example, with “name” implemented as a reference instead of a subdocument:
  - joeNameID = db.names.find({“first”: “Joe”, “last”: “Schmoe”})
    db.people.find({“name” : joeNameID})

Department of Computer Science

PURDUE UNIVERSITY

Queries on References

- Instead of manually resolving references, the $lookup operation can be used to join documents
- 4 fields in $lookup:
  - from: the collection in the *same* database to perform the join with – cannot be a sharded collection
  - localField: the field in the local collection to match with
  - foreginField: the field in the foreign collection to match with
  - as: the name of the new array field to add to the input documents
- $lookup can be used in conjunction with the *aggregate pipeline* in order to perform additional operations

PURDUE UNIVERSITY

Update

- Three update methods:
  - `db.collection.updateOne(<filter>, <update>, <options>)`
  - `db.collection.updateMany(<filter>, <update>, <options>)`
  - `db.collection.replaceOne(<filter>, <update>, <options>)`
- The method accepts:
  - Selection criteria to determine which documents to update
  - Update criteria, namely how the selected document need to be modified
  - Update options, including upsert that creates a new document if no existing document matches the filter
  - For the updateOne method, if multiple documents are selected by the filter, only one will be modified
  - The replaceOne method replaces a single document within the collection using the replacement document
- Operations performed by an update are atomic within a single document.

Department of Computer Science

PURDUE UNIVERSITY

Update

```
db.users.update(                    ← collection
   { age: { $gt: 18 } },            ← update criteria
   { $set: { status: "A" } },       ← update action
   { multi: true }                  ← update option
)
```

Equivalent to

```
UPDATE users          ← table
SET    status = 'A'   ← update action
WHERE  age > 18       ← update criteria
```

PURDUE UNIVERSITY

Update

- By default, the *db.collection.update()* method updates a single document.
- However, when the *multi* option is specified, it modifies all documents in a collection that match a query.
- The db.collection.update() method either updates specific fields in the existing document or replaces the document
  1. If the <update> document contains update operator modifiers, such as those using the $set modifier, then:
     - The update() method updates only the corresponding fields in the document
     - To update an embedded document or an array as a whole, specify the replacement value for the field.
  2. If the <update> document contains only field:value expressions, then:
     - The update() method replaces the matching document with the <update> document.
     - The update() method does not replace the _id value.

Department of Computer Science

PURDUE UNIVERSITY

Update

- For example, suppose we are making major changes to the following user document:

  {"_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
  "name" : "joe","friends" : 32,"enemies" : 2}

- We want to move the "friends" and "enemies" fields to a "relationships" subdocument. We can change the structure of the document in the shell and then replace the database's version with an update:

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends, "enemies" : joe.enemies};
{
        "friends" : 32,
        "enemies" : 2
}> joe.username = joe.name;
> delete joe.friends;
> delete joe.enemies;
> delete joe.name;
> db.users.replaceOne({"name" : "joe"}, joe);
```

Structure of document after the update

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "username" : "joe",
    "relationships" : {
        "friends" : 32,
        "enemies" : 2
    }
}
```

PURDUE UNIVERSITY

Update field operators

Name	Description
$inc	Increments the value of the field by the specified amount.
$mul	Multiplies the value of the field by the specified amount.
$rename	Renames a field.
$setOnInsert	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
$set	Sets the value of a field in a document.
$unset	Removes the specified field from a document.
$min	Only updates the field if the specified value is less than the existing field value.
$max	Only updates the field if the specified value is greater than the existing field value.
$currentDate	Sets the value of a field to current date, either as a Date or a Timestamp.

Department of Computer Science

PURDUE UNIVERSITY

Update with Upsert

- If the update() method includes *upsert: true*
  - And no document matches the selection criteria, a new document is created.
  - And at least one document matches the criteria, the matching documents are modified
- *upsert: true* specifies that, if no matching documents are found for the update, an insert need to be executed.

PURDUE UNIVERSITY

Delete

- db.collection.delete() permanently delete documents from the collection
- The db.collection.delete() method accepts a query criteria to determine which documents to remove
- Called with no parameters, it removes all documents from a collection

E.g.

> db.blog.delete({title : "My Blog Post"})

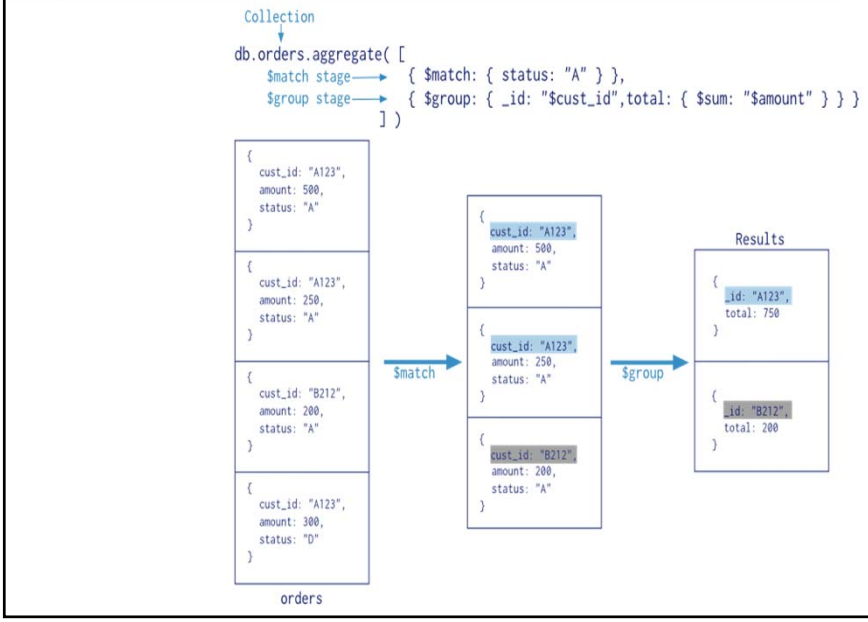
- As with the updates, the remove method has two forms:
  - db.collection.deleteMany
  - Db.collection.deleteOne

PURDUE UNIVERSITY

Aggregation Pipeline

- A framework for data aggregation based on data processing pipelines.
  - Documents enter a multi-stage pipeline that transforms them
- Pipeline stages do not need to produce one output document for every input document. Stages can:
  - generate new documents
  - filter out documents
- MongoDB provides the db.collection.aggregate() method in the mongo shell
  - Pipeline stages appear in an array. Documents pass through the stages in sequence. Stages can appear multiple times in a pipeline.

    >db.collection.aggregate([{ <stage> }, ...])



Main Aggregation Stage Operators

Select the documents to pass unmodified into the next pipeline stage.

$match — For each input document, outputs either one document (a match) or zero documents (no match).

Reshapes each document in the stream

$project — E.g., add new fields or remove existing fields.
For each input document, outputs one document.

PURDUE UNIVERSITY

Main Aggregation Stage Operators

$unwind	*Deconstructs an array field from the input documents to output a document for each element.* Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements (n=0 for an empty array).
$group	*Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group.* Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.

Department of Computer Science

PURDUE UNIVERSITY

Main Aggregation Stage Operators

$sort	*Reorders the document stream by a specified sort key*. The documents remain unmodified. For each input document, outputs one document.
$limit	*Passes the first n documents unmodified to the pipeline where n is the specified limit*. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents)
$skip	*Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline*. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).

PURDUE UNIVERSITY

$match

- Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

Syntax: *{ $match: { <query> } }*

E.g., suppose to have the collection

```
{ "_id" : ObjectId("512bc95fe835e68f199c8686"),
        "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("55f5a192d4bede9ac365b257"),
        "author" : "ahn", "score" : 60, "views" : 1000 }

>db.articles.aggregate(
   [ { $match : { author : "dave" } } ]
);
```


PURDUE UNIVERSITY

$project

Reshapes each document in the stream

Syntax: *{ $project: { <specifications> } }*

<field>: <1 or true>	Specifies the inclusion of a field. if field does not exist no inclusion is performed
_id: <0 or false>	Specifies the suppression of the _id field. only achievable with _id
<field>: <expression>	Adds a new field or reset the value of an existing field.

PURDUE UNIVERSITY

$project

```
{
  "_id" : 1,
  title: "abc123",
  isbn: "0001122223334",
  author: { last: "zzz", first:
          "aaa" },
  copies: 5
}
```

```
>db.books.aggregate(
 [{$project: {
        title: 1,
        _id: 0,
        isbn: {
          prefix: { $substr: [ "$isbn", 0, 3 ] },
          group: { $substr: [ "$isbn", 3, 2 ] },
          publisher: { $substr: [ "$isbn", 5, 4 ] },
          title: { $substr: [ "$isbn", 9, 3 ] },
          checkDigit: { $substr: [ "$isbn", 12, 1] }
        },
        lastName: "$author.last",
        copiesSold: "$copies"
 }}])
```

We can see this aggregation pipeline as the creation of a temporary view of a given document.
Unlike views in the relational DBMS, here a view may change the structure of the returned document.

Department of Computer Science

PURDUE UNIVERSITY

$unwind

- Deconstructs an array field from the input documents to output a document for each element.

- Prototype form: *{ $unwind: <field path> }*

E.g., collection inventory includes the document
{ "_id" : 1, "item" : "ABC1", sizes: ["S", "M", "L"] }

```
> db.inventory.aggregate( [ { $unwind : "$sizes" } ] )
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```

$group

- Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping.
- Prototype:
  { $group:
  { _id: <expression>, <field1>:
  { <accumulator1> : <expression1> }, ... } }
  - The _id field specifies the mandatory grouping key
  - the <accumulator> specifies the aggregation operation

>{"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}}

Department of Computer Science
PURDUE UNIVERSITY

Main Accumulator Operators

$sum	Returns a sum for each group. Ignores non-numeric values.
$avg	Returns an average for each group. Ignores non-numeric values.
$first/$last	Returns a value from the first/last document for each group. Order is only defined if the documents are in a defined order.
$min/$max	Returns the lowest/highest expression value for each group.
$push	Returns an array of expression values for each group.
$addToSet	Returns an array of unique expressions values for each group. Order of the array elements is undefined.

PURDUE UNIVERSITY

$group Example

{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }

Calculate total price and average quantity by date

```
>db.sales.aggregate( [{$group : {
        _id : { month: { $month: "$date" },
                day: {    $dayOfMonth: "$date" },
                year: { $year: "$date" } },
        totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
        averageQuantity: { $avg: "$quantity" }}}])
```

Department of Computer Science

PURDUE UNIVERSITY

MongoDB Shell

- A JavaScript shell that allows interaction with a MongoDB instance from the command line
- Useful for:
  - Performing administrative functions
  - Inspecting a running instance
  - Executing queries
- To start the shell, run the *mongo* executable:

```
$ mongo
MongoDB shell version: 3.0
connecting to: test
>
```

- The shell is a full-featured JavaScript interpreter, capable of running arbitrary JavaScript
- On startup, the shell connects to the *test* database on a MongoDB server and assigns this database connection to the global variable *db*
- *db* is the primary access point to your MongoDB server through the shell
- CRUD operations: create, read, update, and delete to manipulate and view data in the shell

PURDUE UNIVERSITY

Other Document DBMS

Apache CouchDB relax

OrientDB

RAVENDB

terrastore

PostgreSQL

amazon DynamoDB

MS Azure DocumentDB

Ranked list: http://db-engines.com/en/ranking/document+store