NoSQL Databases

Introduction to MongoDB

DANIELE APILETTI

POLITECNICO DI TORINO

Querying data – find() operation

Query language

- Most of the operations available in SQL language can be expressend in MongoDB language

MySQL clause	MongoDB operator
`SELECT`	`find()`

SELECT *
FROM people
	db.people.**find()**

Read data from documents

- Select documents

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

Read data from documents (Filter conditions)

- Select documents

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<conditions>` are optional
    - conditions take a document with the form:

      ```
      {field1 : <value>, field2 : <value> ... }
      ```

    - Conditions may specify a value or a regular expression

Read data from documents (Project fields)

- Select documents

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<fields of interest>` are optional
    - projections take a document with the form:

      ```
      {field1 : <value>, field2 : <value> ... }
      ```

    - **1**/true to include the field, **0**/false to exclude the field

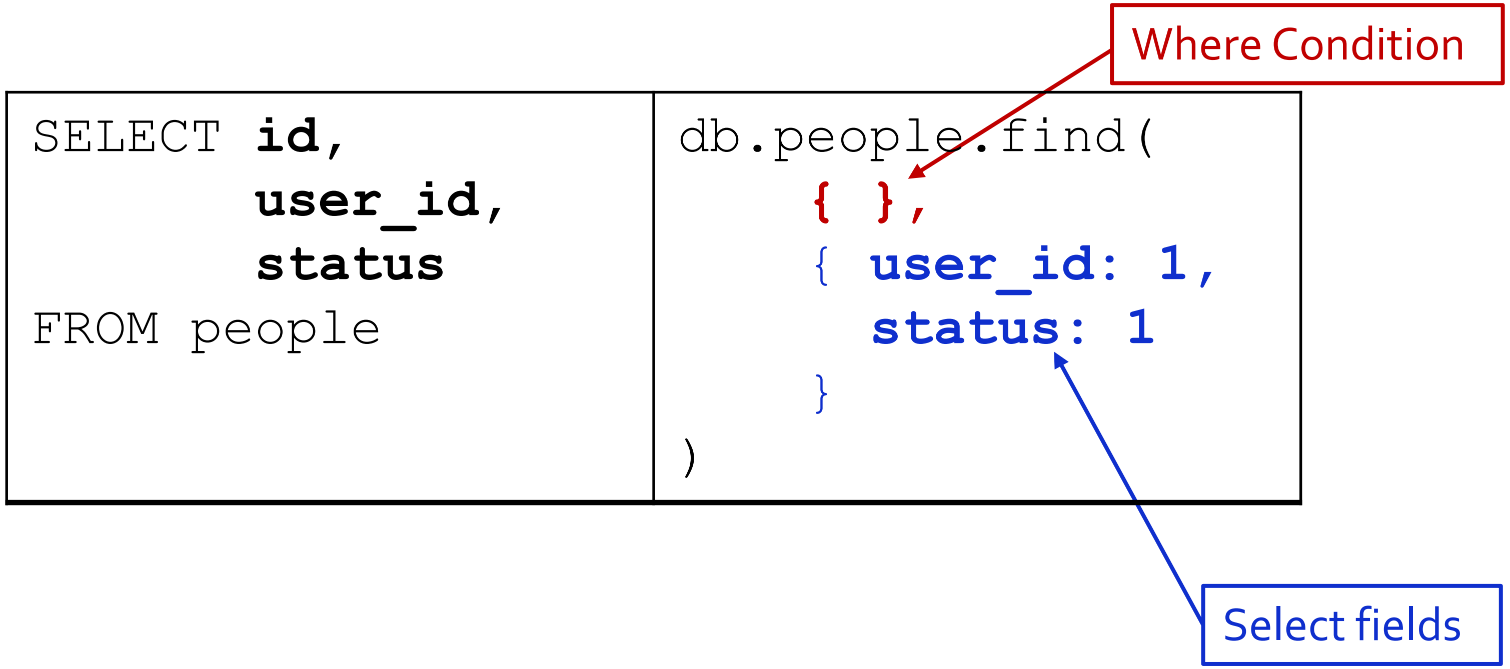
find() operator (1)

```
SELECT id,
       user_id,
       status
FROM people
```

```
db.people.find(
    { },
    { user_id: 1,
      status: 1
    }
)
```

find() operator (2)

MySQL clause	MongoDB operator
`SELECT`	`find()`



```
SELECT id,
       user_id,
       status
FROM people
```

```
db.people.find(
      { },
      { user_id: 1,
        status: 1
      }
)
```

Where Condition

Select fields

find() operator (3)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

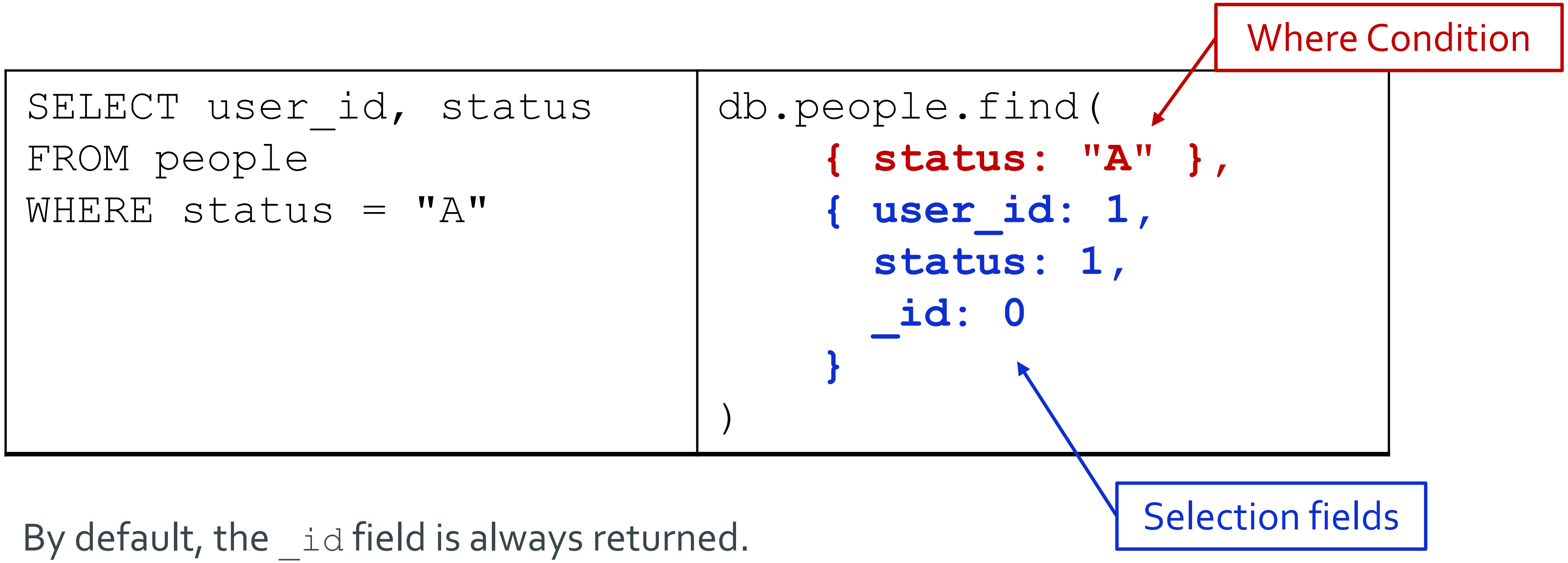
```
SELECT *
FROM people
WHERE status = "A"
```

```
db.people.find(
    { status: "A" }
)
```

Where Condition

find() operator (4)

MySQL clause	MongoDB operator
`SELECT`	`find()`
`WHERE`	`find({<WHERE CONDITIONS>})`



```
SELECT user_id, status
FROM people
WHERE status = "A"
```

```
db.people.find(
     { status: "A" },
     { user_id: 1,
       status: 1,
       _id: 0
     }
)
```

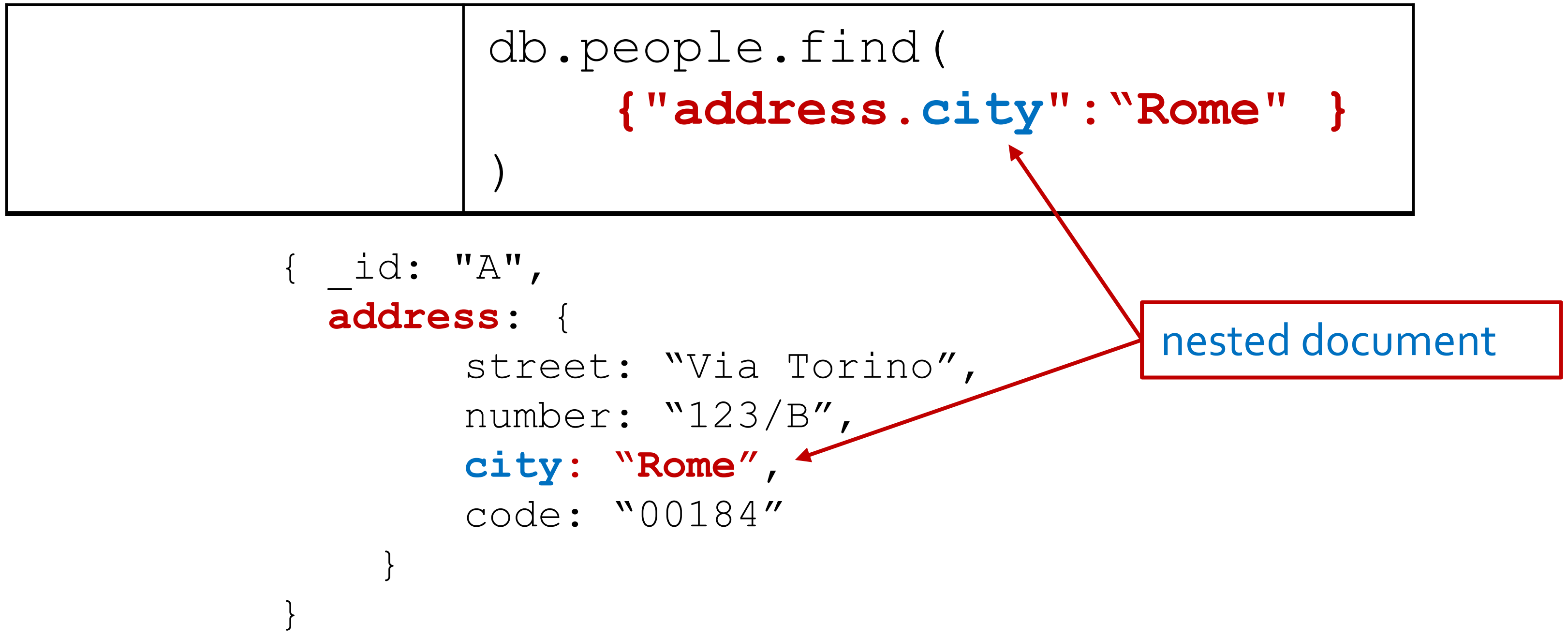
Where Condition

Selection fields

By default, the `_id` field is always returned.
To remove it, you must explicitly indicate `_id: 0`

find() operator (5)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})



```
db.people.find(
     {"address.city":“Rome"  }
)
```

```
{ _id: "A",
  address: {
        street: “Via Torino”,
        number: “123/B”,
        city: “Rome”,
        code: “00184”
     }
}
```

nested document

Read data from one document

- Select a single document

```
db.<collection name>.findOne( {<conditions>}, {<fields of interest>} );
```

- Select one document that satisfies the specified query criteria.
    - If multiple documents satisfy the query, it returns the first one according to the natural order which reflects the order of documents on the disk.

(No) joins

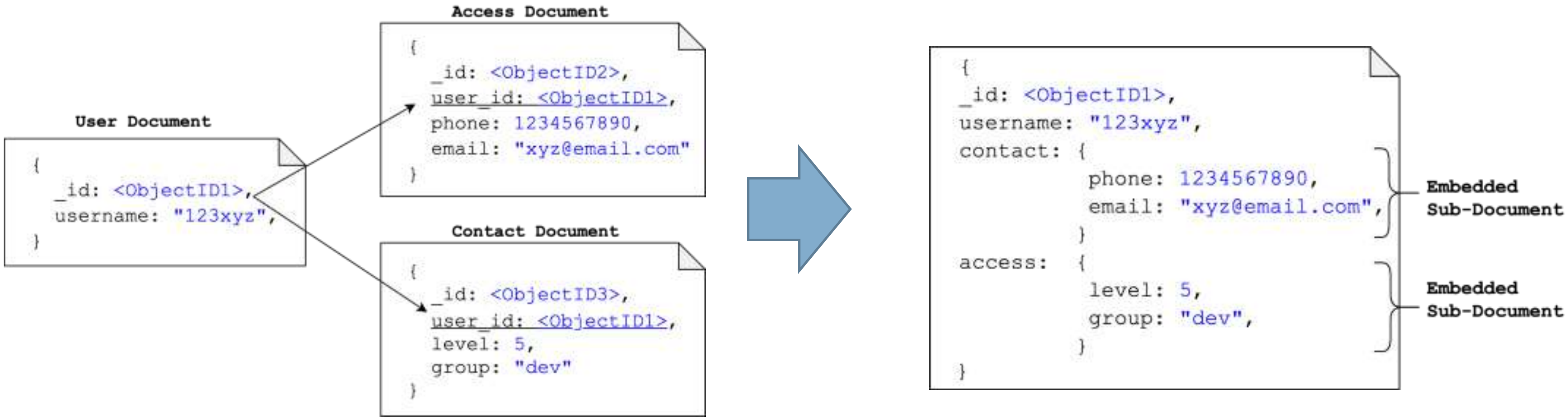
- No join operator exists (but `$lookup`)
  - You must write a program that
    - Selects the documents of the first collection you are interested in
    - Iterates over the documents returned by the first step, by using the loop statement provided by the programming language you are using
    - Executes one query for each of them to retrieve the corresponding document(s) in the other collection

https://docs.mongodb.com/manual/reference/operator/aggregation/lookup

(No) joins

- **(no) joins**
  - Relations among documents/records are provided by
    - Object_ID (_id), named “**Manual reference**” in MongoDB, a second query is required
    - **DBRef**, a standard approach across collections and databases (check the driver compatibility)

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```



User Document

```
{
   _id: <ObjectID1>,
   username: "123xyz",
}
```

Access Document

```
{
   _id: <ObjectID2>,
   user_id: <ObjectID1>,
   phone: 1234567890,
   email: "xyz@email.com"
}
```

Contact Document

```
{
   _id: <ObjectID3>,
   user_id: <ObjectID1>,
   level: 5,
   group: "dev"
}
```

```
{
_id: <ObjectID1>,
username: "123xyz",
contact: {
            phone: 1234567890,
            email: "xyz@email.com",
          }
access:   {
            level: 5,
            group: "dev",
          }
}
```

https://docs.mongodb.com/manual/reference/database-references/

Comparison query operators

Name	Description
`$eq or   :`	Matches values that are equal to a specified value
`$gt`	Matches values that are greater than a specified value
`$gte`	Matches values that are greater than or equal to a specified value
`$in`	Matches any of the values specified in an array
`$lt`	Matches values that are less than a specified value
`$lte`	Matches values that are less than or equal to a specified value
`$ne`	Matches all values that are not equal to a specified value, **including documents that do not contain the field**.
`$nin`	Matches none of the values specified in an array

Comparison operators (>)

MySQL	MongoDB	Description
>	**$gt**	**greater than**

SELECT *
FROM people
WHERE **age > 25**
	db.people.find(
	   { **age: { $gt: 25 }** }
	)

Comparison operators (>=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	**$gte**	**greater equal then**

```
SELECT *
FROM people
WHERE age >= 25
```

```
db.people.find(
   { age: { $gte: 25 } }
)
```

Comparison operators (<)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	**$lt**	**less than**

```
SELECT *
FROM people
WHERE age < 25
```

```
db.people.find(
   { age: { $lt: 25 } }
)
```

Comparison operators (<=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	**$lte**	**less equal then**

SELECT *
FROM people
WHERE **age <= 25**
	db.people.find(
	   { **age: { $lte: 25 }** }
	)

Comparison operators (=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	**$eq**	**equal to** The $eq expression is equivalent to { field: <value> }.

```
SELECT *
FROM people
WHERE age = 25
```

```
db.people.find(
   { age: { $eq: 25 } }
)
```

Comparison operators (!=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	$eq	equal to
!=	**$ne**	**Not equal to**

```
SELECT *
FROM people
WHERE age != 25
```

```
db.people.find(
   { age: { $ne: 25 } }
)
```

Conditional operators

- To specify multiple conditions, **conditional operators** are used
- MongoDB offers the same functionalities of MySQL with a different syntax.

MySQL	**MongoDB**	**Description**
AND	,	Both verified
OR	**$or**	At least one verified

Conditional operators (AND)

MySQL	MongoDB	Description
AND	,	**Both verified**

```
SELECT *
FROM people
WHERE status = "A"
AND age = 50
```

```
db.people.find(
    { status: "A",
      age: 50 }
)
```

Conditional operators (OR)

MySQL	MongoDB	Description
AND	,	Both verified
OR	**$or**	**At least one verified**

```
SELECT *
FROM people
WHERE status = "A"
OR age = 50
```

```
db.people.find(
{  $or:
   [ { status: "A" } ,
     { age: 50 }
   ]
}
)
```

Type of read operations (1)

- Count

```
db.people. count({ age: 32 })
```

- Comparison

```
db.people. find({ age: {$gt: 32 })              // or equivalently with $gte, $lt, $lte,
```

```
db.people.find({ age: {$in: [32, 40] })          // returns all documents having age either 32 or 40
```

```
db.people.find({ age: { $gt: 25, $lte: 50 } })   //returns all documents having age > 25 and age <= 50
```

- Logical

```
db.people.find({ name: {$not: {$eq: “Max” } } })
```

```
db.people.find({ $or: [ {age: 32}, {age: 33} ] } )
```

Type of read operations (2)

```
db.items.find({
    $and: [
        {$or: [{qty: {$lt: 15}}, {qty: {$gt: 50}} ]},
        {$or: [{sale: true}, {price: {$lt: 5}} ]}
    ]
```

This query returns documents (items) that satisfy **both** these conditions:

1. Quantity sold either less than 15 **or** greater than 50
2. Either the item is on sale (field “sale”: true) **or** its price is less than 5

Type of read operations (3)

- Element

```
db.inventory.find( { item: null } )                        // equality filter
```

```
db.inventory.find( { item : { $exists: false } } )         // existence filter
```

```
db.inventory.find( { item : { $type: 10 } } )              // type filter
```

Note:

- Item: null → matches documents that either
  - contain the item field whose value is **null** or
  - that do **not** contain the item field
- Item: {$exists: false} → matches documents that do **not** contain the item field

- Aggregation → Slides on “Data aggregation”

Type of read operations (4)

- Embedded Documents

```
db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
```

Select all documents where the field size equals the **exact document** { h: 14, w: 21, uom: "cm" }

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

To specify a query condition on fields in an embedded/nested document, use **dot notation**

```
db.inventory.find( { "size.h": { $lt: 15 } } )
```

Dot notation and comparison operator

Type of read operations (5)

- Array
  - Query for all documents where the field tags value is an array with exactly two specific elements

```
db.inventory.find( { tags: ["red", "black"] } )                → Item list order matters!
db.inventory.find( { tags: { $all: ["red", "black"] } } )      → List order does not matter
```

- The following queries return **different** results, i.e., they are **not** equivalent

```
db.inventory.find( { tags: ["red", "black"] } )
db.inventory.find( { tags: ["black", "red"] } )
```

```
db.inventory.find( { tags: { $all: ["red", "black"] } } )
db.inventory.find( { tags: { $all: ["black", "red"] } } )
```

Type of read operations (6)

- Query for all documents where tags is an array that **contains** the string "red" as one of its elements

```
db.inventory.find( { tags: "red" } )
```

- Query an Array with **Compound Filter Conditions** on the Array Elements

```
db.inventory.find( { dim_cm: { $gt: 15, $lt: 20 } } )
```

- Query for an Array Element that **Meets Multiple Criteria**

```
db.inventory.find( { dim_cm: { $elemMatch: { $gt: 15, $lt: 20 } } } )
```

Attention:

- **Compound filter**: one element of the array can satisfy the greater than 15 condition and another element can satisfy the less than 20 condition, or alternatively a single element can satisfy both
- **elemMatch**: one single element of the array **must** satisfy **both**

Type of read operations (7)

- Query for an Element by the Array Index Position

```
db.inventory.find( { "dim_cm.0": { $gt: 25 } } )
```

- Query an Array by Array Length

```
db.inventory.find( { "tags": { $size: 3 } } )
```

Cursor

- `db.collection.find()` gives back a cursor. It can be used to iterate over the result or as input for next operations.
- E.g.,
  - `cursor.sort()`
  - `cursor.count()`
  - `cursor.forEach() //shell method`
  - `cursor.limit()`
  - `cursor.max()`
  - `cursor.min()`
  - `cursor.pretty()`

Cursor: sorting data

- Sort is a cursor method
- Sort documents
  - `sort( {<list of field:value pairs>} );`
  - field specifies which filed is used to sort the returned documents
  - value = -1 descending order
  - Value = 1 ascending order
- Multiple field: value pairs can be specified
  - Documents are sort based on the first field
  - In case of ties, the second specified field is considered

Cursor: sorting data

- Sorting data with respect to a given field in sort() operator

MySQL clause	MongoDB operator
`ORDER BY`	`sort()`

`SELECT *`
`FROM people`
`WHERE status = "A"`
`ORDER BY age ASC`
	`db.people.find(`
	`  { status: "A" }`
	`).`**`sort( { age: 1 } )`**

- Returns all documents having status=“A”. The result is sorted in ascending age order

Cursor: sorting data

- Sorting data with respect to a given field in sort() operator

MySQL clause	**MongoDB operator**
`ORDER BY`	`sort()`

`SELECT *`
`FROM people`
`WHERE status = "A"`
`ORDER BY age ASC`
	`db.people.find(`
	`  { status: "A" }`
	`).`**`sort( { age: 1 } )`**

`SELECT *`
`FROM people`
`WHERE status = "A"`
`ORDER BY age DESC`
	`db.people.find(`
	`  { status: "A" }`
	`).`**`sort( { age: -1 } )`**

- Returns all documents having status=“A”. The result is sorted in ascending age order
- Returns all documents having status = “A”. The result is sorted in descending age order

Cursor: counting

MySQL clause	MongoDB operator
COUNT	count()or find().count()

SELECT COUNT(*)
FROM people
	db.people.count()
	or
	db.people.find().count()

Cursor: counting

MySQL clause	MongoDB operator
COUNT	count()or find().count()

SELECT COUNT(*)
FROM people
	db.people.count()
	or
	db.people.find().count()

SELECT COUNT(*)
WHERE status = "A"
FROM people
	db.people.count(status: "A")}
	or
	db.people.find({status: "A"}).count()

Cursor: counting

MySQL clause	**MongoDB operator**
COUNT	count()or find().count()

SELECT COUNT(*)
FROM people
	db.people.count()
	or
	db.people.find().count()

SELECT COUNT(*)
WHERE status = "A"
FROM people
	db.people.count(status: "A")}
	or
	db.people.find({status: "A"}).count()

SELECT COUNT(*)
FROM people
WHERE **age > 30**
	db.people.count(
	{ age: { $gt: 30 } }
	)

Similar to the find() operator, count() can embed conditional statements.

Cursor: forEach()

- `forEach` applies a JavaScript function to apply to each document from the cursor.

```
    db.people.find({status: "A“}).forEach(
        function(myDoc){
                print( "user:”+myDoc.name );
        })
```

- Select documents with status=“A” and print the document name.

Databases and collections.

Update operations (3)

Document update

- Back at the C.R.U.D. operations, we can now see how documents can be updated using:

```
db.collection.updateOne(<filter>, <update>, <options>)
```

```
db.collection.updateMany(<filter>, <update>, <options>)
```

  - `<filter>` = filter condition. It specifies which documents must be updated
  - `<update>` = specifies which fields must be updated and their new values
  - `<options>` = specific update options

Document update

- E.g.,

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

  - This operation updates all documents with qty<50
  - It sets the value of the size.uom field to "in", the value of the status field to "P", and the value of the lastModified field to the current date.

Updating data

- Tuples to be updated should be selected using the WHERE statements

MySQL clause	MongoDB operator

UPDATE <table>
SET <statement>
WHERE <condition>
	db.<table>.updateMany(
	   { <condition> },
	   { $set: {<statement>} }
	)

Updating data

MySQL clause	MongoDB operator

UPDATE <table>
SET <statement>
WHERE <condition>
	db.<table>.updateMany(
	    { <condition> },
	    { $set: {<statement>}}
	)

UPDATE people
SET status = "C"
WHERE age > 25
	db.people.updateMany(
	    {age: { $gt: 25 } },
	    {$set: { status: "C"}}
	)

Updating data

MySQL clause	MongoDB operator

UPDATE <table>
SET <statement>
WHERE <condition>
	db.<table>.updateMany(
	    { <condition> },
	    { $set: {<statement>}}
	)

UPDATE people
SET status = "C"
WHERE age > 25
	db.people.updateMany(
	    {age: { $gt: 25 } },
	    {$set: { status: "C"}}
	)

UPDATE people
SET age = **age + 3**
WHERE status = "A"
	db.people.updateMany(
	    { status: "A" } ,
	    { **$inc: { age: 3 }** }
	)

The $inc operator increments a field by a specified value

Data aggregation

General concepts

- Documents enter a multi-stage pipeline that transforms the **documents of a collection** into an aggregated result
- Pipeline **stages** can appear **multiple** times in the pipeline
  - exceptions *$out*, *$merge*, and *$geoNear* stages
- Pipeline expressions can **only** operate on the **current document** in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents (max 100 Mb of RAM per stage).
- Generally, expressions are **stateless** and are only evaluated when seen by the aggregation process with one exception: accumulator expressions used in the *$group* stage (e.g. totals, maximums, minimums, and related data).
- The aggregation pipeline provides an alternative to ***map-reduce*** and may be the preferred solution for aggregation tasks since MongoDB introduced the *$accumulator* and *$function* aggregation operators starting in version 4.4

Aggregation Framework

SQL	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
//LIMIT	$limit
SUM	$sum
COUNT	$sum

Aggregation pipeline

- Aggregate functions can be applied to collections to group documents

```
db.collection.aggregate( { <set of stages> })
```

  - Common stages: `$match, $group ..`
  - The aggregate function allows applying aggregating functions (e.g. sum, average, ..)
  - It can be combined with an initial definition of groups based on the grouping fields

Aggregation example (1)

```
db.people.aggregate( [
   { $group: { _id: null,
               mytotal: { $sum: "$age" },
               mycount: { $sum: 1 }
         }
   }
] )
```

- Considers all documents of people and
  - sum the values of their age
  - sum a set of ones (one for each document)
- The returned value is associated with a field called “mytotal” and a field “mycount”

Aggregation example (2)

```
db.people.aggregate( [
   { $group: { _id: null,
            myaverage: { $avg: "$age" },
            mytotal: { $sum: "$age" }
         }
   }
] )
```

- Considers all documents of people and computes
  - sum of age
  - average of age

Aggregation example (3)

```
db.people.aggregate( [
    { $match: {status: "A"} } ,
    { $group: { _id: null,
                count: { $sum: 1 }
            }
    }
] )
```

Where conditions

- Counts the number of documents in people with status equal to “A”

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       AVG(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $avg: "$age" }
     }
   }
] )
```

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
    {
      $group: {
         _id: "$status",
         total: { $sum: "$age" }
      }
    }
] )
```

Group field

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
    {
      $group: {
         _id: "$status",
         total: { $sum: "$age" }
      }
    }
] )
```

Group field

Aggregation function

Aggregation in “Group By + Having”

MySQL clause	MongoDB operator
`HAVING`	`aggregate($group, $match)`

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Aggregation in “Group By + Having”

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Group stage: Specify the aggregation field and the aggregation function

Aggregation in “Group By + Having”

MySQL clause	MongoDB operator
`HAVING`	`aggregate($group, $match)`

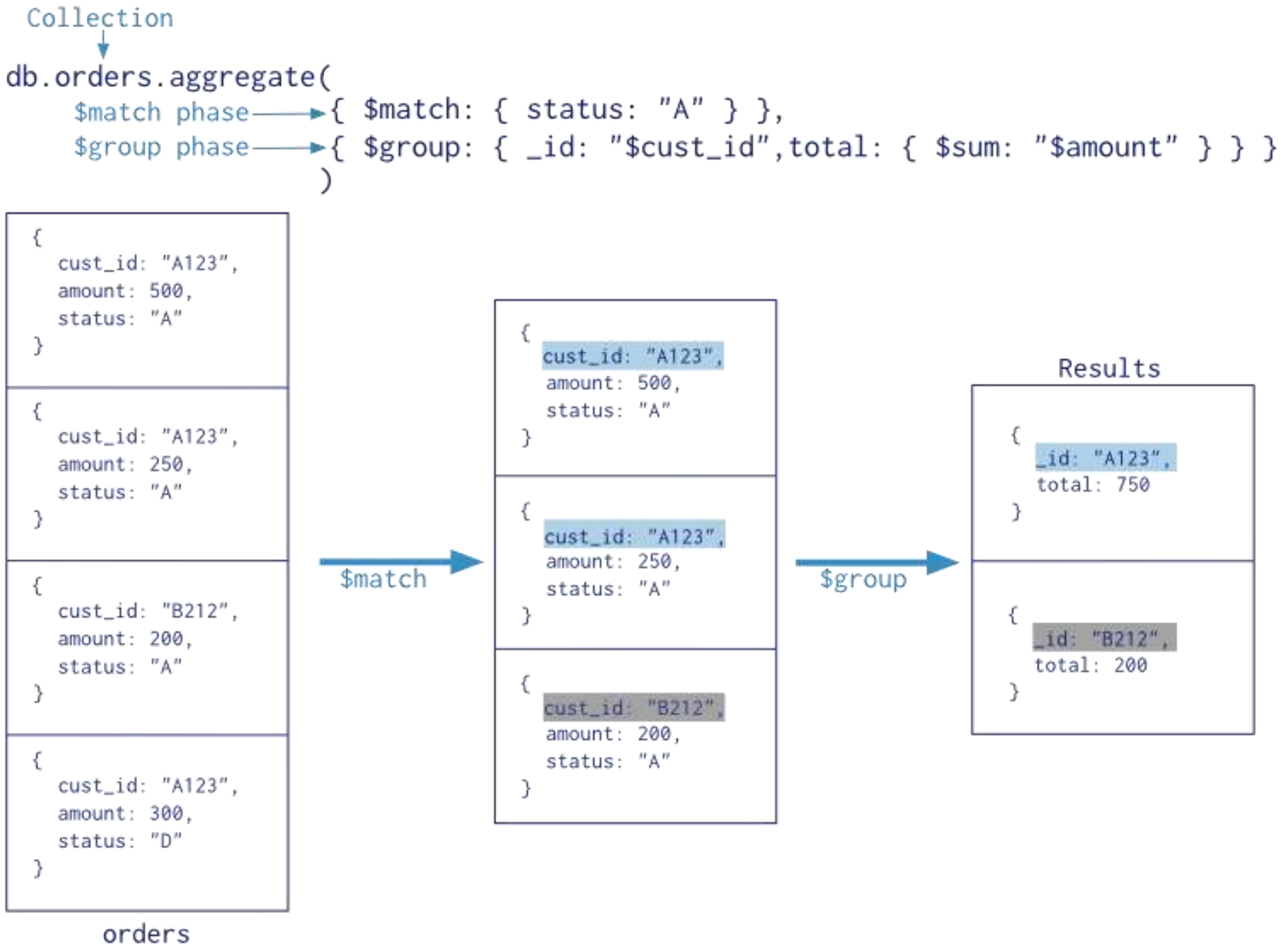
```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
   { $match: { total: { $gt: 1000 } } }
] )
```

Group stage: Specify the aggregation field and the aggregation function

Match Stage: specify the condition as in `HAVING`

Aggregation at a glance



```
db.orders.aggregate(
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
)
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{
  _id: "A123",
  total: 750
}
{
  _id: "B212",
  total: 200
}
```

Pipeline stages (1)

Stage	Description
$addFields	Adds **new fields** to documents. Reshapes each document by adding new fields to output documents that will contain both the existing fields from the input documents and the newly added fields.
$bucket	Categorizes incoming documents **into groups**, called buckets, based on a specified expression and bucket boundaries. On the contrary, **$group** creates a “bucket” for each value of the group field.
$bucketAuto	Categorizes incoming documents into a specific number of groups, called buckets, based on a specified expression. Bucket boundaries are automatically determined in an attempt to **evenly distribute** the documents into the specified number of buckets.
$collStats	Returns statistics regarding a collection or view (it must be the **first stage**)
$count	Passes a document to the next stage that contains a count of the input **number of documents** to the stage (same as $group+$project)

Pipeline stages (2)

Stage	**Description**
$facet	Processes **multiple aggregation pipelines** within a single stage on the same set of input documents. Enables the creation of multi-faceted aggregations capable of characterizing data across multiple dimensions. Input documents are passed to the $facet stage only once, without needing multiple retrieval.
$geoNear	Returns an ordered stream of documents based on the **proximity** to a geospatial point. The output documents include an additional **distance** field. It must in the **first stage** only.
$graphLookup	Performs a **recursive search** on a collection. To each output document, adds a new array field that contains the traversal results of the recursive search for that document.

Example

```
db.employees.aggregate( [
  {
    $graphLookup: {
      from: "employees",
      startWith: "$reportsTo",
      connectFromField: "reportsTo",
      connectToField: "name",
      as: "reportingHierarchy"
    }
  }
] )
```

- The $graphLookup operation recursively matches on the **reportsTo** and **name** fields in the employees collection, returning the **reporting hierarchy** for each person.
- Returns a list of documents such as

```
{
  "_id" : 5,
  "name" : "Asya",
  "reportsTo" : "Ron",
  "reportingHierarchy" : [
    { "_id" : 1, "name" : "Dev" },
    { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
    { "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" }
  ]
}
```

original document

Pipeline stages (3)

Stage	Description
$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.
$indexStats	Returns statistics regarding the use of each index for the collection.
$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).
$lookup	Performs a **join** to another collection in the same database to filter in documents from the “joined” collection for processing. To each input document, the $lookup stage adds a new array field whose elements are the matching documents from the “joined” collection. The $lookup stage passes these reshaped documents to the next stage.

Pipeline stages (4)

Stage	Description
$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$merge	Writes the resulting documents of the aggregation pipeline to a collection. The stage can incorporate (insert new documents, merge documents, replace documents, keep existing documents, fail the operation, process documents with a custom update pipeline) the results into an output collection. To use the $merge stage, it must be the last stage in the pipeline.
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out stage, it must be the last stage in the pipeline.
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

Pipeline stages (5)

Stage	Description
$sample	Randomly selects the specified number of documents from its input.
$set	Adds new fields to documents. Similar to $project, $set reshapes each document in the stream; specifically, by adding new fields to output documents that contain both the existing fields from the input documents and the newly added fields. $set is an alias for $**addFields** stage. If the name of the new field is the same as an existing field name (including _id), $set **overwrites** the existing value of that field with the value of the specified expression.
$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).
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.

Pipeline stages (6)

Stage	Description
$sortByCount	Groups incoming documents based on the value of a specified expression, then computes the count of documents in each distinct group.
$unset	Removes/excludes fields from documents.
$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 and can be zero for an empty array.