BD2 NOSQL - MONGO

SERGIO ÁLVAREZ

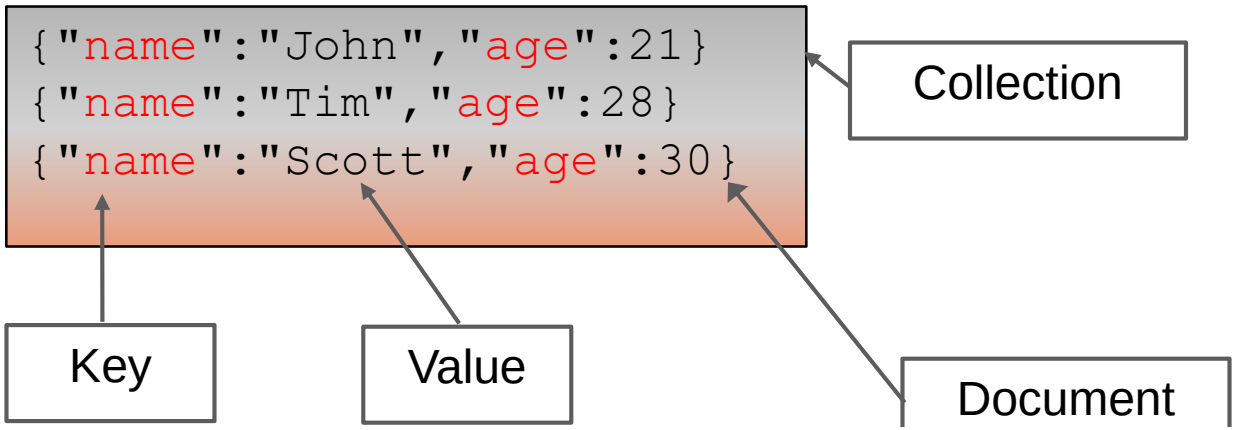
VERSIÓN 1.0

JSON

```
{
“Key”:”Value”
"string":"John",
"number":123.45,
"boolean":true,
"array":[ "a", "b", "c" ],
"object": { "str":"Miller", "num":711 },
"value": NULL,
"date": ISODate("2013-10-01T00:33:14.000Z")
}
```

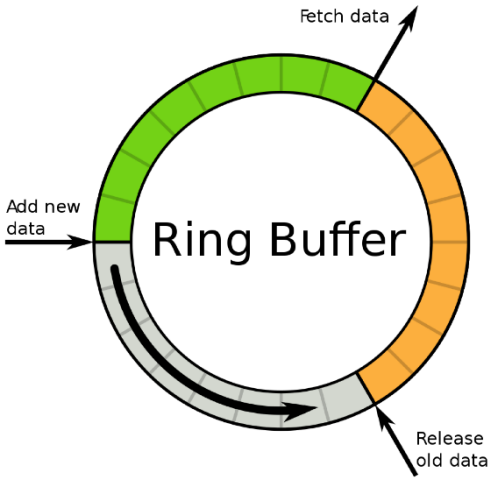
TRANSACCIONES Y CONVENCIONES

Name	Age
John	21
Tim	28
Scott	30



CAPPED COLLECTIONS

Una colección "capped" es un tipo especial de colección que tiene un tamaño fijo o un número fijo de elementos. Una vez que la colección está "llena", los elementos más viejos se eliminan cada vez que añadimos nuevos.



USE CUSTOM COLLATION

Field	Type	Description
`locale`	string	The ICU locale. See Supported Languages and Locales for a list of supported locales. To specify simple binary comparison, specify `locale` value of `"simple"`.
`strength`	integer	Optional. The level of comparison to perform. Corresponds to ICU Comparison Levels. Possible values are:

Se realiza un índice que no distingue entre mayúsculas y minúsculas al especificar una intercalación con una intensidad de 1 o 2. Puede crear un índice insensible a mayúsculas y minúsculas como este:

```
db.myCollection.createIndex({city: 1}, {collation: {locale: "en", strength: 2}});
```

VALIDACIONES

SCHEMA VALIDATION

validationAction	Description
"error"	**Default** Documents must pass validation before the write occurs. Otherwise, the write operation fails.
"warn"	Documents do not have to pass validation. If the document fails validation, the write operation logs the validation failure.

validationLevel	Description
"off"	No validation for inserts or updates.
"strict"	**Default** Apply validation rules to all inserts and all updates.
"moderate"	Apply validation rules to inserts and to updates on existing *valid* documents. Do not apply rules to updates on existing *invalid* documents.

SCHEMA VALIDATION

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "name", "year", "major", "address" ],
         properties: {
            name: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            year: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "must be an integer in [ 2017, 3017 ] and is required"
            },
            major: {
               enum: [ "Math", "English", "Computer Science", "History", null ],
               description: "can only be one of the enum values and is required"
            },
            gpa: {
               bsonType: [ "double" ],
               description: "must be a double if the field e
            },
         validationAction: "warn"
```

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

MOTORES Y VERSIÓN ENTERPRICE

STORAGE ENGINES

The storage engine is the component of the database that is responsible for managing how data is stored, both in memory and on disk. MongoDB supports multiple storage engines, as different engines perform better for specific workloads. Choosing the appropriate storage engine for your use case can significantly impact the performance of your applications.

> NOTE:
> Starting in version 4.2, MongoDB removes the deprecated MMAPv1 storage engine.

➤ **WiredTiger Storage Engine (*Default*)**

WiredTiger is the default storage engine starting in MongoDB 3.2. It is well-suited for most workloads and is recommended for new deployments. WiredTiger provides a document-level concurrency model, checkpointing, and compression, among other features.

In MongoDB Enterprise, WiredTiger also supports Encryption at Rest. See Encrypted Storage Engine.

➤ **In-Memory Storage Engine**

In-Memory Storage Engine is available in MongoDB Enterprise. Rather than storing documents on-disk, it retains them in-memory for more predictable data latencies.

Encrypted Storage Engine

New in version 3.2.

> ENTERPRISE FEATURE:
> Available in MongoDB Enterprise only.

> IMPORTANT:
> Available for the WiredTiger Storage Engine only.

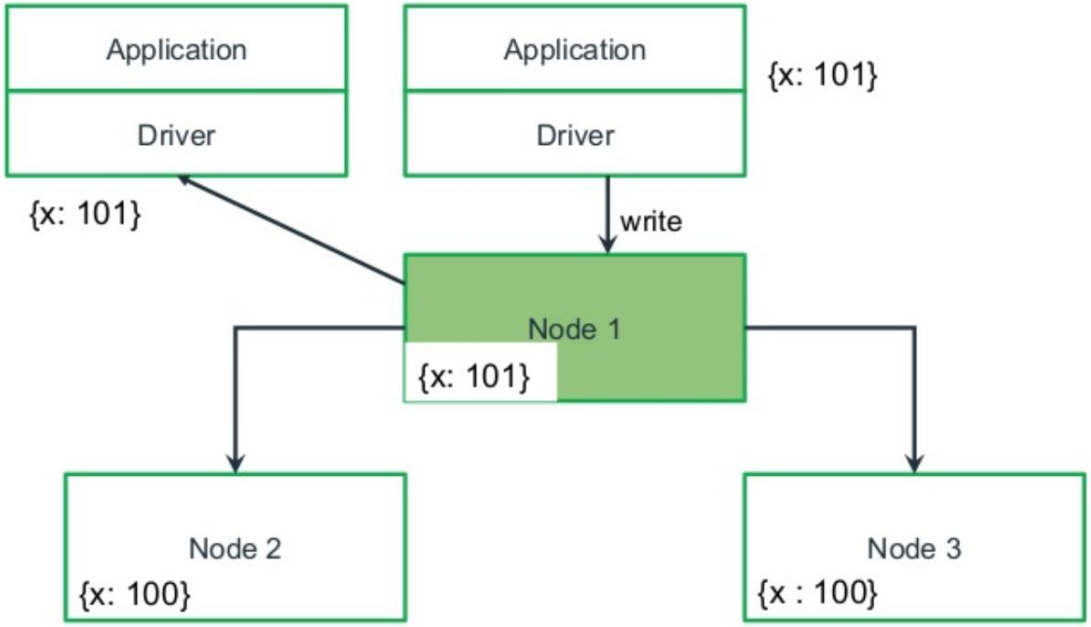
MongoDB Enterprise 3.2 introduces a native encryption option for the WiredTiger storage engine. This feature allows MongoDB to encrypt data files such that only parties with the decryption key can decode and read the data.

NIVEL DEL CONSISTENCIA

LECTURA LOCAL

ReadConcern local

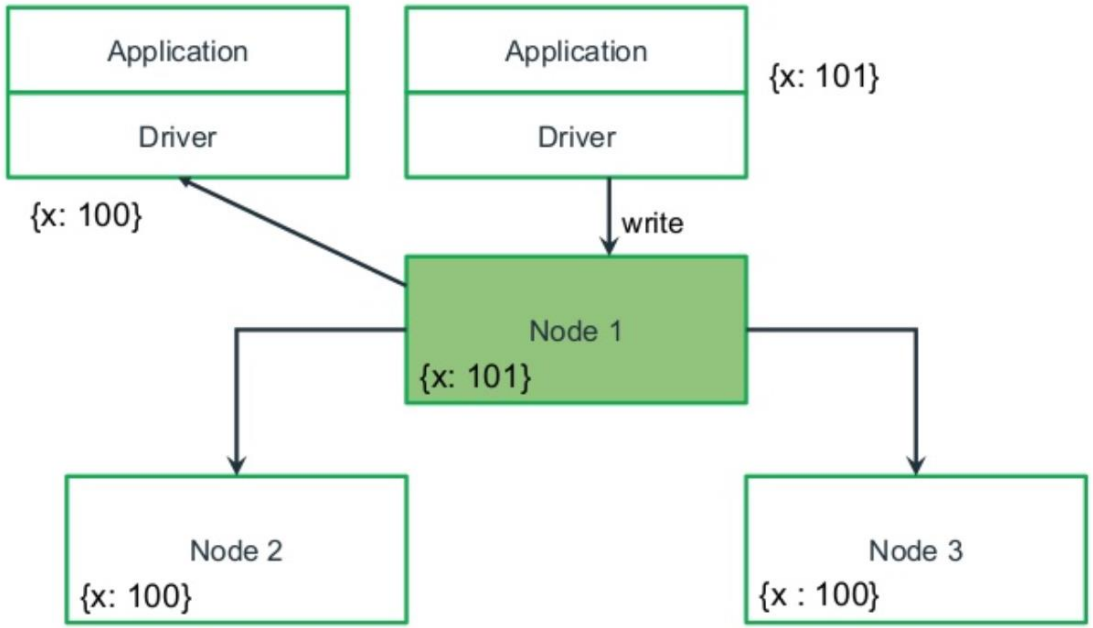
```
db.test.find({x : {$gt : 100}})
    .readConcern("local")
```



LECTURA MAYORÍA

ReadConcern: Majority

```
db.test.find({x : {$gt : 100}})
    .readConcern("majority")
```



ESCRITURAS

Write Concern

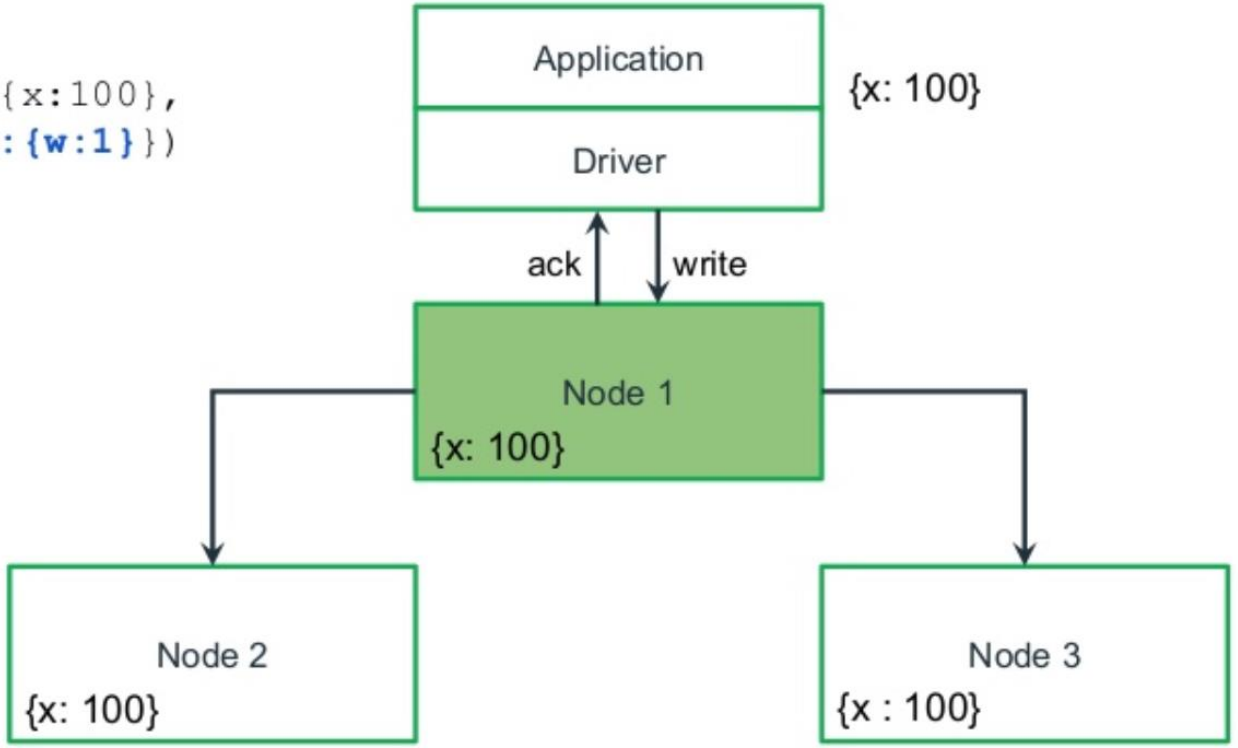
- Intelligent write receipt/confirmation
  - Specifies the the number of nodes that must have written the write to disk
  - Default number is 1

```
db.test.insert({x:100},{writeConcern:{w:2}})
```

ESCRITURA EN 1

Write Concern 1

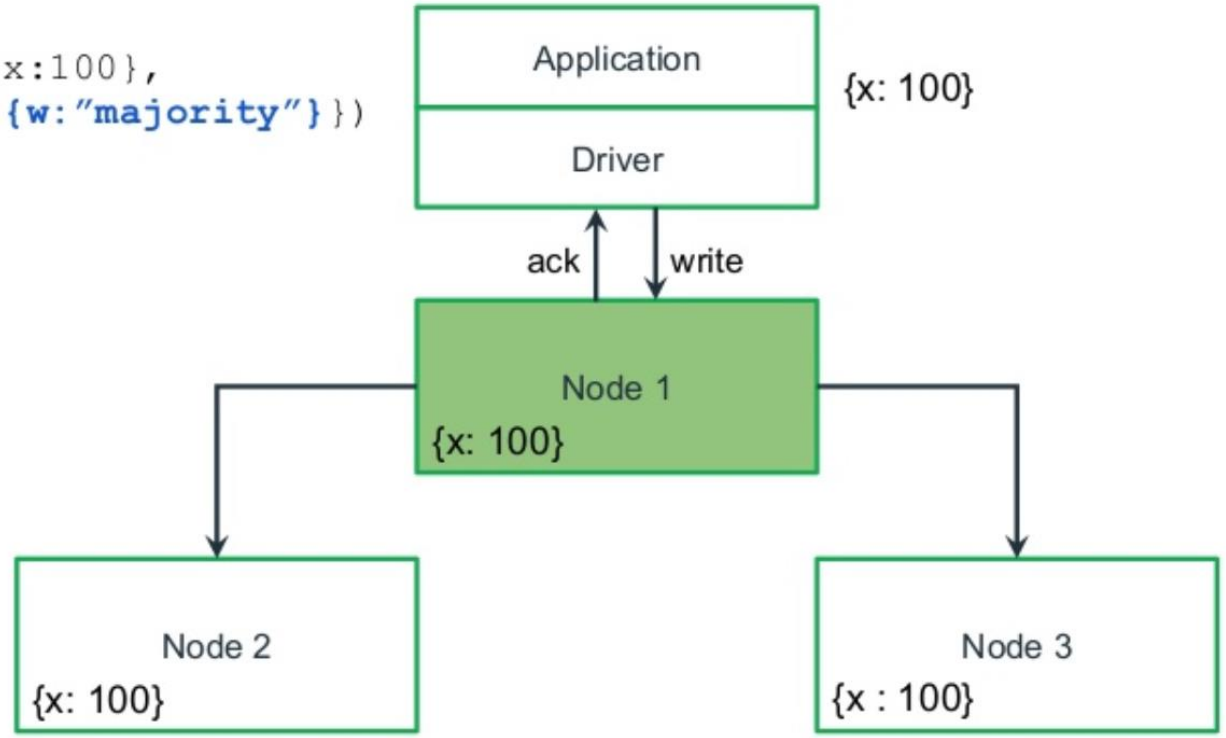
```
db.test.insert({x:100},
  {writeConcern:{w:1}})
```



ESCRITURA EN MAYORIA

Write Concern Majority

```
db.test.insert({x:100},
  {writeConcern:{w:"majority"}})
```



DOES MONGODB SUPPORT TRANSACTIONS?

Because a single document can contain related data that would otherwise be modeled across separate parent-child tables in a relational schema, MongoDB's atomic single-document operations already provide transaction semantics that meet the data integrity needs of the majority of applications. One or more fields may be written in a single operation, including updates to multiple sub-documents and elements of an array. The guarantees provided by MongoDB ensure complete isolation as a document is updated; any errors cause the operation to roll back so that clients receive a consistent view of the document.

However, for situations that require atomicity of reads and writes to multiple documents (in a single or multiple collections), MongoDB supports multi-document transactions:

- **In version 4.0**, MongoDB supports multi-document transactions on replica sets.
- **In version 4.2**, MongoDB introduces distributed transactions, which adds support for multi-document transactions on sharded clusters and incorporates the existing support for multi-document transactions on replica sets.

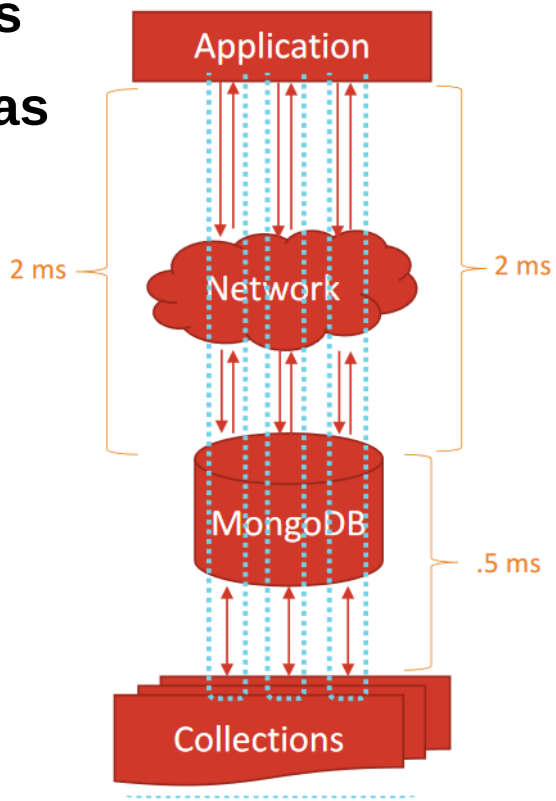
TRANSACCIONES

```
client = MongoClient(uriString)
wc_majority = WriteConcern("majority", wtimeout=1000)

# Prereq: Create collections.
client.get_database(
    "mydb1", write_concern=wc_majority).foo.insert_one({'abc': 0})
client.get_database(
    "mydb2", write_concern=wc_majority).bar.insert_one({'xyz': 0})

# Step 1: Define the callback that specifies the sequence of operations to perform inside
def callback(session):
    collection_one = session.client.mydb1.foo
    collection_two = session.client.mydb2.bar

    # Important:: You must pass the session to the operations.
    collection_one.insert_one({'abc': 1}, session=session)
    collection_two.insert_one({'xyz': 999}, session=session)

# Step 2: Start a client session.
with client.start_session() as session:
    # Step 3: Use with_transaction to start a transaction, execute the callback, and commi
    session.with_transaction(
        callback, read_concern=ReadConcern('local'),
        write_concern=wc_majority,
        read_preference=ReadPreference.PRIMARY)
```

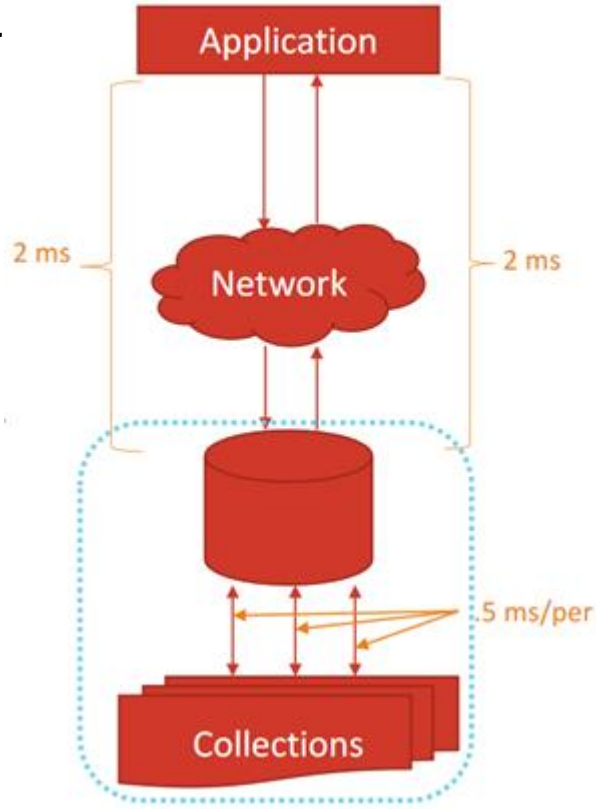
AGGREGATION-PIPELINE

PORQUE USAR PIPELINES

Múltiples Consultas



Consulta Pipeline



SQL TO AGGREGATION MAPPING CHART (1/2)

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum $sortByCount
join	$lookup
SELECT INTO NEW_TABLE	$out
MERGE INTO TABLE	$merge (Available starting in MongoDB 4.2)
UNION ALL	$unionWith (Available starting in MongoDB 4.4)

SQL TO AGGREGATION MAPPING CHART (2/2)

https://docs.mongodb.com/manual/reference/sql-aggregation-comparison/

copy

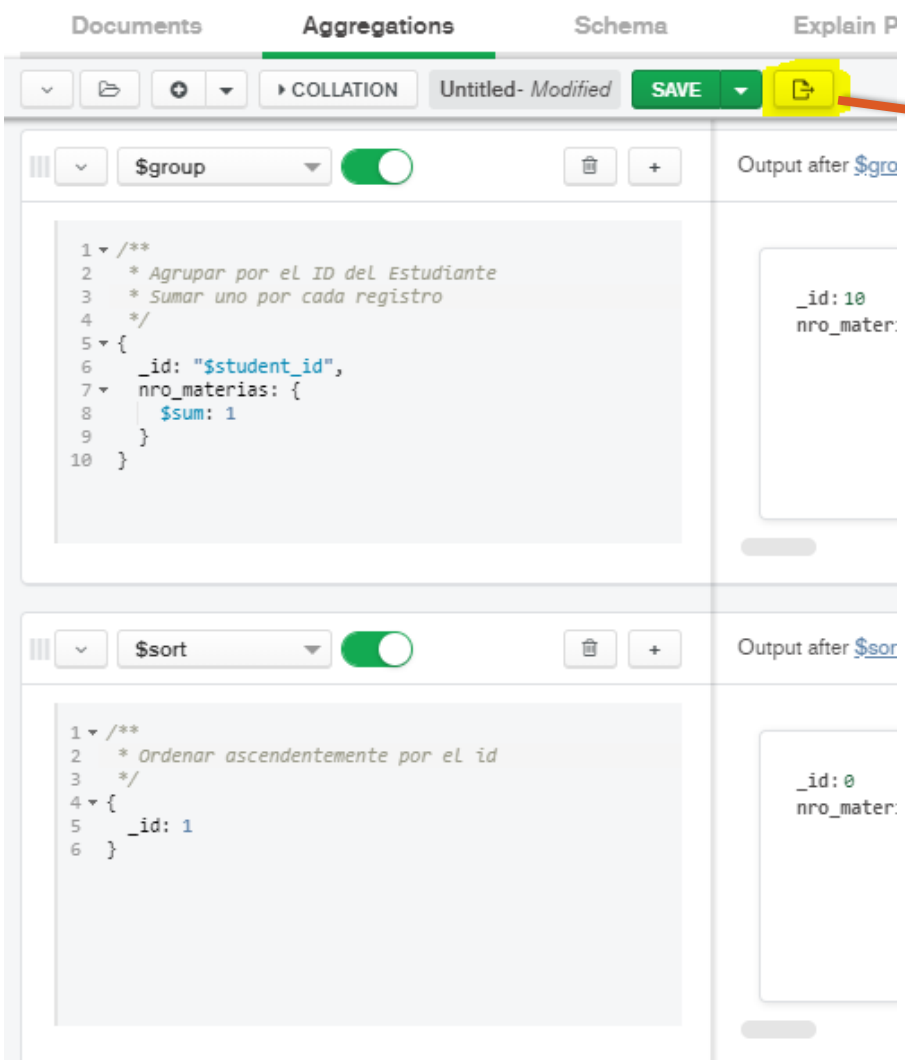
```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

SQL Example	MongoDB Example	Description
`SELECT COUNT(*) AS count FROM orders`	`db.orders.aggregate( [ { $group: { _id: null, count: { $sum: 1 } } } ] )`	Count all records from `orders`

STAGE (1/2)	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.
$lookup	Performs a left outer join to another collection in the *same* database to filter in documents from the “joined” collection for processing.
$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).
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

STAGE (2/2)	DESCRIPTION
$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.
$replaceWith	Replaces a document with the specified embedded document. The operation replaces all existing fields in the input document, including the _id field. Specify a document embedded in the input document to promote the embedded document to the top level.
$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.

CONSULTAR NÚMERO DE MATERIAS POR ESTUDIANTE



My Pipeline:

```
[{
    $group: {
        _id: "$student_id",
        nro_materias: {
            $sum: 1
        }
    }
}, {
    $sort: {
        _id: 1
    }
}]
```

USO VALIDACIONES EN COLECCIONES

JSON SCHEMA

ejemplo_ciudades.personas

Documents | Aggregations | Schema | Explain Plan | Indexes | **Validation**

Validation Action ERROR ▾ Validation Level STRICT ▾

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'nombre1',
      'apellido1',
      'ciudad_id_nace'
    ],
    properties: {
      nombre1: {
        bsonType: 'string',
        pattern: '^[a-zA-Z]{3,}$',
        description: 'El primer nombre de la requerido es requer
      },
      nombre2: {
        bsonType: 'string',
        pattern: '^[a-zA-Z]{3,}$',
        description: 'El segundo nombre es carcater de longitud minima 3'
      },
      ciudad_id_nace: {
        bsonType: 'int',
        description: 'ID ciudad id'
      },
      apellido1: {
        bsonType: 'string',
        pattern: '^[a-zA-Z]{3,}$',
        description: 'El primer apellido de la requerido es requerido y es carcater de longitud minima 3'
      },
```

Que hacer

Campos Mandatorios (obligatorios)

Especificación de un campo
Tipo de dato, Expresión regular y Documentación

https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/#op._S_jsonSchema

INSERTAR DOCUMENTOS

Insert to Collection ejemplo_ciudades.personas

VIEW {} ≡

```
1 ▾ {
2     "nombre1":"pedro",
3     "apellido1":"galindo"
4   }
```

Document failed validation

```
required: [
  'nombre1',
  'apellido1',
  'ciudad_id_nace'
],
properties: {
  nombre1: {
    bsonType: 'string',
    pattern: '^[a-zA-Z]{3,}$',
    description: 'El primer nom
  },
  nombre2: {
    bsonType: 'string',
    pattern: '^[a-zA-Z]{3,}$',
    description: 'El segundo no
  },
  ciudad_id_nace: {
    bsonType: 'int',
    description: 'ID ciudad id'
  },
```

Insert to Collection ejemplo_ciudades.personas

VIEW {} ≡

```
1 ▾ {
2       "nombre1": "pedro",
3       "apellido1": "galindo",
4       "ciudad_id_nace": 3,
5       "telefono": 1234567
6   }
7
```

ejemplo_ciudades.personas

Documents Aggregations

FILTER {"apellido1":"galindo"}

ADD DATA VIEW {} ≡

```
_id: ObjectId("5f6a363e08f5fa1538a8edc5")
nombre1: "pedro"
apellido1: "galindo"
ciudad_id_nace: 3
telefono: 1234567
```

ANIDANDO VALIDACIONES

ejemplo_ciudades.personas

Documents Aggregations

```
34
35    ciudades_vive: {
36      bsonType: ['array'],
37      required: [
38        'ciudad_id',
39        'feinicio',
40        'fefin'
41      ],
42      properties: {
43        ciudad_id: {
44          bsonType: 'int',
45          description: 'ID ciudad'
46        },
47        feinicio: {
48          bsonType: 'date',
49          description: 'fecha inicio'
50        },
51        fefin: {
52          bsonType: 'date',
53          description: 'fecha fin'
54        }
55      }
56    }
```

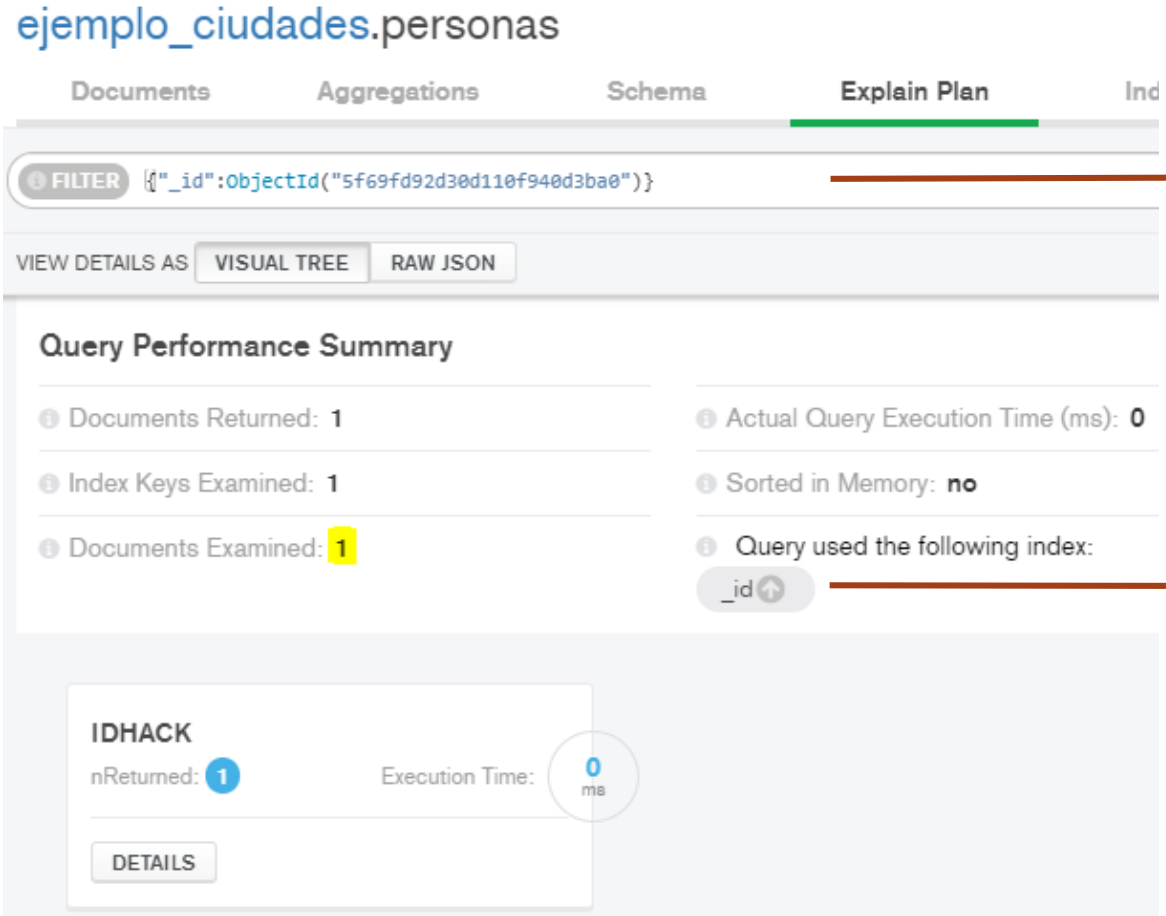
A nivel global el “ARREGLO” ciudades_vive es opcional

Pero si se usa, se debe componen de forma obligatoria de 3 campos

Especificación de un campo
Tipo de dato y Documentación

OPTIMIZACIÓN

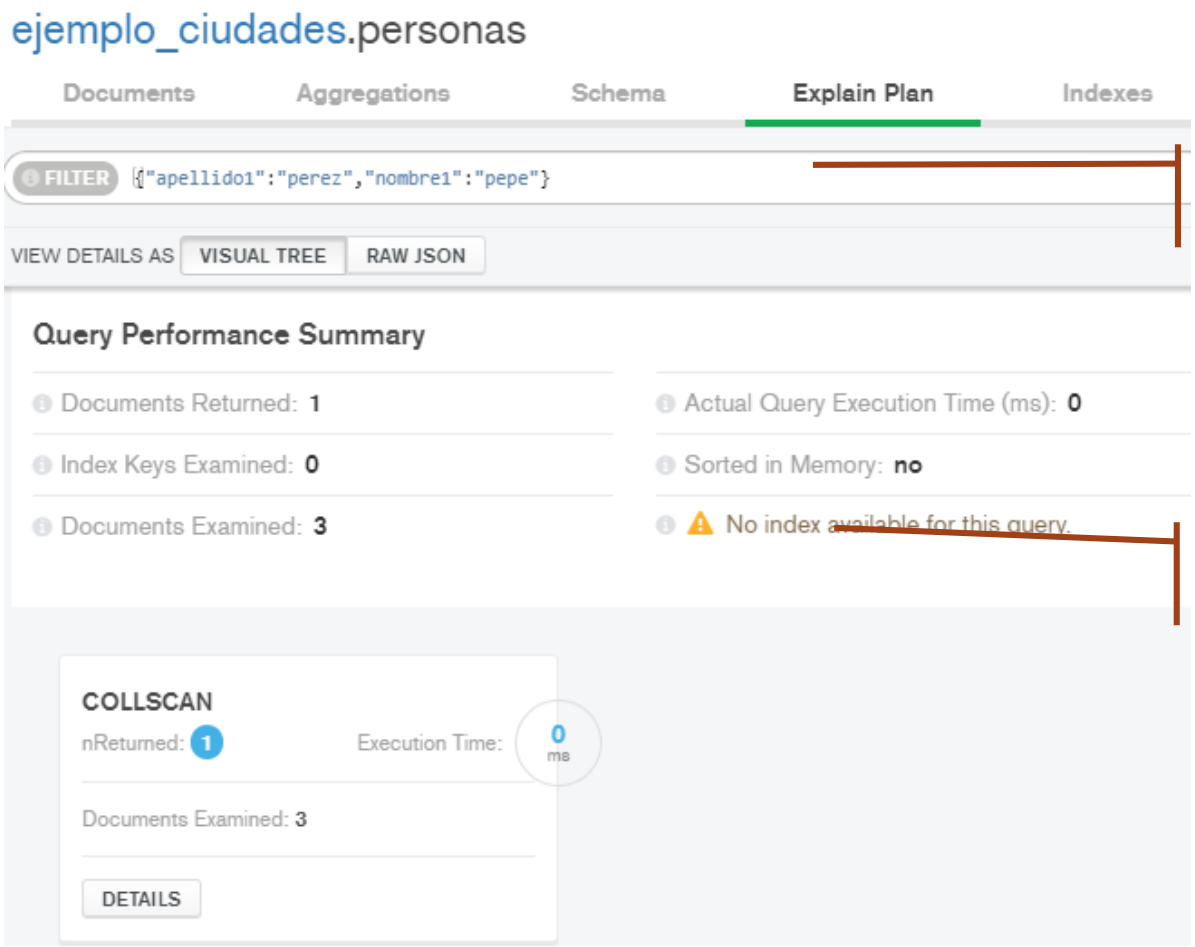
EXPLAIN PLAN (1/2)



Consulta

Uso Indice?

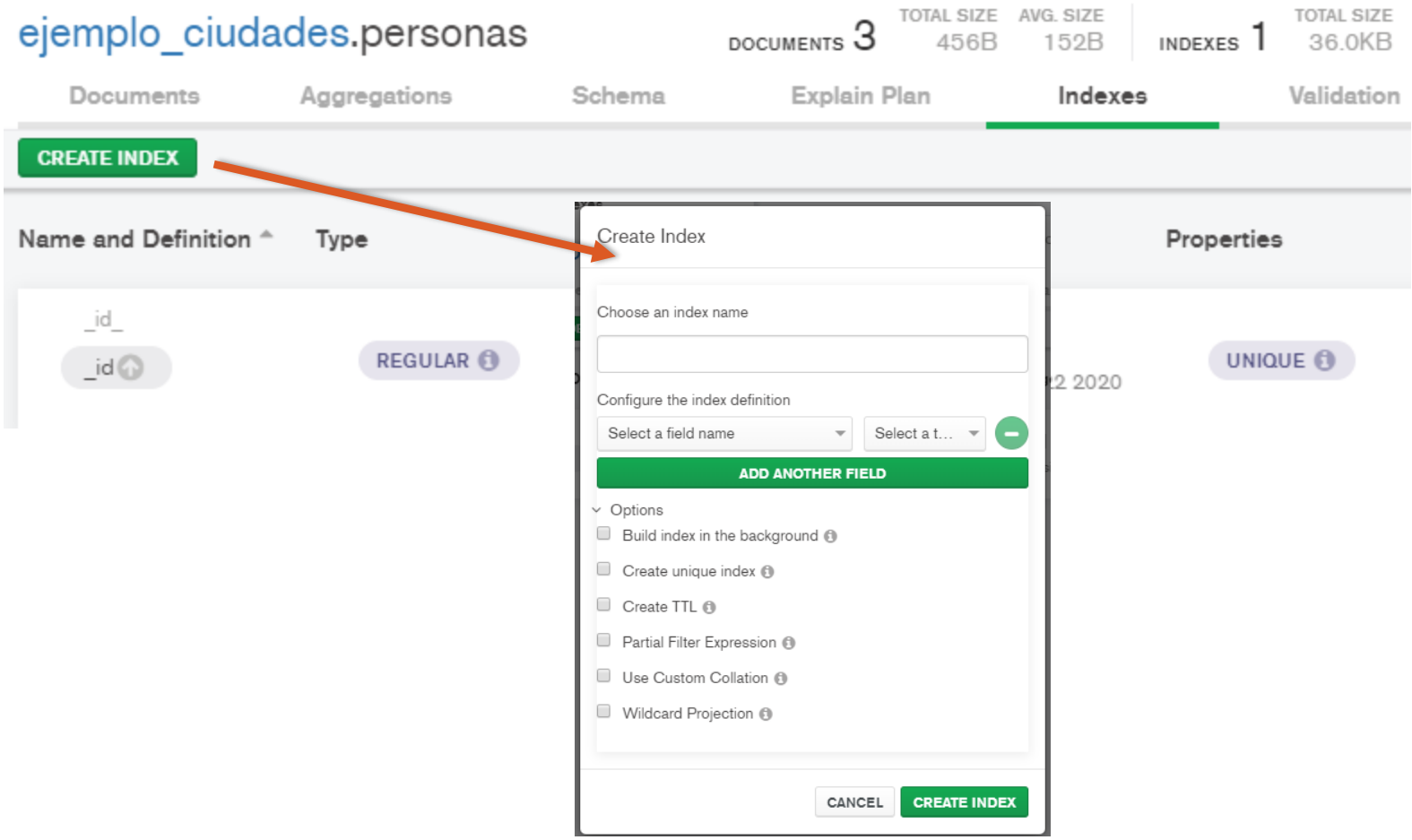
EXPLAIN PLAN (2/2)



Consulta

Uso Indice?

ÍNDICES



https://docs.mongodb.com/manual/indexes/#index-types

ÍNDICE COMPUESTO

ejemplo_ciudades.personas

Documents | Aggregations | Schema | Explain Pla

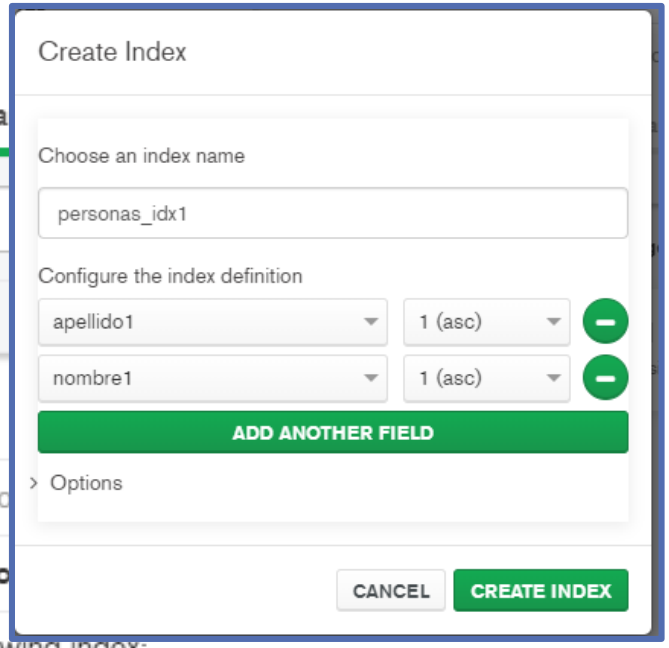
FILTER {"apellido1":"perez","nombre1":"pepe"}

VIEW DETAILS AS VISUAL TREE | RAW JSON

Query Performance Summary

- Documents Returned: 1
- Index Keys Examined: 1
- Documents Examined: 1
- Actual Query Executic
- Sorted in Memory: no
- Query used the following index:

apellido1 ↑ nombre1 ↑



FILTER {"apellido1":"perez"} SI

FILTER {"nombre1":"pepe"} NO