- Introduction
  - What
    - Common Features
    - Provides
  - Why
    - Supports
  - Types of NoSQL
    - 1. Key-Value Stores
      - Example
    - 2. Column Family / Columnar Databases
      - Use Cases
    - 3. Graph Databases
      - Use Cases
    - 4. Document Store Databases
      - Example
- MongoDB
  - The Document Model and MongoDB
    - Structure and Syntax
    - Collections
      - Document
      - Collection
  - MQL: MongoDB Query Language
    - MQL Create
    - MQL Find
      - Query filter document
      - Specifying query operators
        - Comparison Operators
        - Logical Operators
      - Examples
    - MQL Update
    - MQL Delete
    - MQL Drop
- MongoDB Aggregation Expressions
  - Syntax
  - Suggested ordering of stages in the pipeline
  - Stages
    - $match
      - $expr
    - $project
    - $limit
    - $group
    - $addFields
      - $function
    - $unwind
      - Options
  - Examples
    - $lookup

# Introduction

## What

- NoSQL stands for **not only SQL**
- family of databases that vary widely in style and technology

### Common Features

1. Non relational in featured
2. not RDBMS

### Provides

1. New ways of storing and querying data
2. Databases do not require fixed schemas
3. Horizontally scale easily
4. Provide native fault tolerance and availability in distributed systems

## Why

Applications shifting from internal employees to the public internet users requires high availability and performance

### Supports

- flexible data model
- built in horizontal and vertical scaling
- developer productivity
- distributed environments
- **Polyglot persistence**: mixing RDBMS solutions with NoSQL solutions

## Types of NoSQL

Key-Value, Document, Column, Graph

### 1. Key-Value Stores

Simplest NoSQL databases - in which it stores data as a collection of key-vale pairs

1. Keys are unique
2. Directly points to its associated data

#### Example

**Redis or Memcached**: caching frequently used data
**DynamoDB**: E-commerce platforms, gaming applications, high scalable

### 2. Column Family / Columnar Databases

1. Organize data in columns rather than rows
2. Efficient for handling data sets with dynamic schemas

#### Use Cases

1. **Apache Cassandra**: IoT applications
2. **Hbase**: applications that store and analyze user preferences and behaviours
3. Large-scale data analysis

### 3. Graph Databases

Designed to manage highly interconnected data, representing relationships

#### Use Cases

1. Social Networking
2. Recommendation systems

### 4. Document Store Databases

1. Store data in document format (JSON, BSON, or XML)
2. Each document contains key-value pairs or key-document pairs
3. These databases are schema-less, allowing flexibility in data structures within a collection

#### Example

1. MongoDB
2. CouchBase
3. Amazon DocumentDB

# MongoDB

## The Document Model and MongoDB

### Structure and Syntax

1. JSON Notation
2. Field-value pairs are separated by colon
3. Fields must be enclosed within quotation marks
4. Two field-values are separated by commas

### Collections

#### Document

A way to organize and store data as a set of field-value pairs in MongoDB

#### Collection

An organized store of documents in MongoDB, usually with common fields between documents

## MQL: MongoDB Query Language

### MQL Create

`insertOne()`: Insert one document into a collection

```
db.cows.insertOne({
        name: "daisy",
        milk: 8
});
```

`insertMany()`: Insert an array of documents into a collection

```
db.people.insertMany([
        {
                "user_id": "Eoin",
                "age": 29,
                "Status": "A"
        },
        {
                "user_id": "Daniel",
                "age": 25,
                "Status": "A",
                "Country": "USA"
        }
], ordered: false);
```

`writeConern`: Sets the level of acknowledgement requested from MongoDB for write operations
`ordered`: For insertMany(), there is an additional option for controlling whether the documents are inserted in ordered or unordered fashion

## MQL Find

### Query filter document

`db.<collection>.find({<field1>:<value>,...})`

### Specifying query operators

`db.<collection>.find({ <field1>: { <operator1>: <value1> }, ... })`

#### Comparison Operators

`$eq`, `$gt`, `$lt`, `$gte`, `$lte`, `$ne`, `$in`, `$nin`

#### Logical Operators

`$and`, `$not`, `$or`, `$nor`

### Examples

#### Using $and

```
// Find cows that have milk between 6 and 9
cowCol.find(
        {
                $and: [
                        {milk: {$gt: 6}},
                        {milk: {$lt: 9}}
                ]
        }
)
```

#### Using implicit AND and OR

```
// Find an inventory that has status "A" and either has a quantity of less than 30 or the item name starts
with a p
dn.inventory.find({
        status: "A",
        {$or: [
                {price: {$lt: 30}},
                {item: /^p/}
        ]}
})
```

#### Using explicit AND and OR

```
// Find an inventory that has (quantity is less than 10 or more than 50) and (status is A or size.w is
less than 15)
dn.inventory.find({
        $and: [
                {$or: [
                        {qty: {$lt: 10}},
                        {qty: {$gt: 50}}
                ]},
                {$or: [
                        {status: "A"},
                        {"size.w": {$lt: 15}}
                ]}
        ]
})
```

#### Using the logical NOT operator

```
// Find items that do not start with e letter p
db.inventory.find({
        item: {$not: {$regex: /^p.*/}}
})
```

## MQL Update

`updateOne()`: Update one document into a collection

```
// Set and change the milk number
db.cows.updateOne(
        {
                name: "daisy",
                milk: 12
        },
        {
                $set: {milk: 8}
        }
);
```

`updateMany()`: Update an array of documents into a collection

```
// Increment all milk count in the collection
db.cows.updateMany(
        {},
        {$inc: {milk, 1}}
);
```

## MQL Delete

`deleteOne()`: Delete one document into a collection

```
// Set and change the milk number
db.cows.deleteOne(
        {
                name: "daisy",
                milk: 9
        }
);
```

`deleteMany()`: Delete an array of documents into a collection

```
// Increment all milk count in the collection
db.cows.deleteMany({});
```

### MQL Drop

`drop()`: Removes the entire database object

```
db.some_collection.drop();
```

# MongoDB Aggregation Expressions

The `$expr` operator allows the use of aggregation expressions in MQL

```
{$expr: {<expression>}}
```

Like functions and take arguments, typically these are an array of arguments.

## Syntax

An aggregation pipeline is an **array** used to hold

1. the **stages** to execute
    - each stage is a document (JSON-like)
2. the **parameters** for that stage
   - the parameters of each stage are stored in the documents
   Conforms to the standard data structure of MongoDB

```
db.cows.aggregate([
        {$match: { ... }},
        {$project: { ... }},
        {$limit: { ... }}
])
```

## Suggested ordering of stages in the pipeline

1. $match - filter documents early
2. $project - reshape documents if needed
3. $unwind - flatten arrays if necessary
4. $group - perform grouping and aggregations
5. $sort - sort the results
6. $limit / $skip - reduce the result set size
7. $lookup - perform joins if needed
8. $facet - perform multi-pipeline operations (if needed)
9. $count or $merge / $out - end the pipeline

## Stages

### $match

This is similar to a SQL WHERE clause. Used to select documents that match certain conditions

```
// get documents that have farm is equal to 1
{$match: {"farm": 1}}
```

```
// We can also use $expr for aggregation expressions. Here we find documents that have a spending greater
than its budget
db.monthlybudget.aggregate([
        {$match:{
                $expr: {
```

```
                        $gt: ["$spent", "$budget"]
                }
        }}
])
```

### $expr

We use `$expr` in match when we need to use calculations or field-to-field comparisons

## $project

Reshapes each document by specifying which fields to include or exclude. In the project stage:
**1** means include the field
**0** means exclude the field
Unspecified fields are excluded by default. However, `_id` is included by default

```
// Reshape the document by only  including the name, milk, but exclude id
{$project: {"name": 1, "milk": 1, "_id": 0}}
```

## $limit

Limits the result set to a specified number. Used when you only need a subset of results

```
// Only want 10
{$limit: 10}
```

## $group

This stage takes the incoming stream of documents, and segments it. Each group is represented by a single document

```
cowcol.aggregate([
        {$group: {
                _id: "$farm",
                total_milk: {$sum: "$milk"}
        }}
])
/*
This results in
{ "_id": 4, "total_milk": 96562 }
{ "_id": 1, "total_milk": 110104 }
{ "_id": 2, "total_milk": 99335 }
{ "_id": 5, "total_milk": 108357 }
{ "_id": 3, "total_milk": 85142 }
*/
```

```
// Find the total marks for each student across all subjects
db.marks.aggregate([
        {$group: {
                _id: "$name",
                "total": {$sum: "$marks"}
        }}
])
```

```
// Find the top two subjects based on average marks
db.mark.aggreagate([
        {$group: {
                _id: $name,
                average: {$avg: $marks}
        }},
        {$sort: {average ,-1}},
```

```
        {$limit: 2}
])
```

## $addFields

This stage takes the incoming stream of documents, and adds a new field to the document as it is processed

### $function

allows the use of custom Javascript functions within an aggregation pipeline. You must declare the

1. body
2. args
3. lang

```
// Add the age field
db.person.aggregate([
        {$addFields: {
                age: {
                        $function: {
                                body: function() {
                                        age = Math.floor((Math.random() * 5) + 1) ;
                                        return age;
                                },
                                args: [],
                                lang: 'js'
                        }
                }
        }}
])
```

## $unwind

Deconstructs an array field outputting a document for each element. It essentially splits a document that contains an array into multiple documents, each with a single value from the array. We need to specify a field path to indicate the array to be deconstructed or specify a document operator.

### Options

1. `preserveNullAndEmptyArrays`: setting this option to true preserves documents that have empty arrays
2. `includeArrayIndex`: this option allows you to include the index of each element in the array in the output

**Sample Document**

```
[
  {
    "_id": 1,
    "name": "Store A",
    "branches": [
      { "locations": ["Downtown", "Uptown"], "employees": 15 },
      { "locations": null, "employees": 8 }
    ]
  },
  {
    "_id": 2,
    "name": "Store B",
    "branches": [
      { "locations": ["Northside"], "employees": 12 },
      { "locations": [], "employees": 20 }
    ]
  },
  {
    "_id": 3,
    "name": "Store C",
    "branches": null
```

```
  }
]
```

### Unwind operation

```
db.stores.aggregate([
        {$unwind: {
                path: "$branches.location",
                preserveNullAndEmptyArrays: true }
        }
])
```

### Result

```
{ "_id": 1, "name": "Store A", "branches": { "locations": "Downtown", "employees": 15 } }
{ "_id": 1, "name": "Store A", "branches": { "locations": "Uptown", "employees": 15 } }
{ "_id": 1, "name": "Store A", "branches": { "locations": null, "employees": 8 } }
{ "_id": 2, "name": "Store B", "branches": { "locations": "Northside", "employees": 12 } }
{ "_id": 2, "name": "Store B", "branches": { "locations": null, "employees": 20 } }
{ "_id": 3, "name": "Store C", "branches": null }
```

## Examples

```
db.budgetitem.aggregate([
  {
    $match: {
      $and: [
        {
          $expr: { $gt: ["$spent", "$budget"] } // Filters documents where 'spent' is greater than
'budget'.
        },
        {
          spent: { $gte: 250 } // Filters documents where 'spent' is greater than or equal to 250.
        },
        {
          "details": { $exists: true } // Ensures the 'details' field exists in the documents.
        }
      ]
    }
  },
  {
    $project: {
      _id: 0, // Excludes the '_id' field from the output.
      expense: 1, // Includes the 'expense' field in the output.
      airline: "$details.airline", // Includes the 'airline' field nested inside 'details'.
      flight_class: "$details.seat_class", // Includes the 'seat_class' field nested inside 'details' as
'flight_class'.
      flight_type: "$details.flight_type", // Includes the 'flight_type' field nested inside 'details'.
      overspend: { $subtract: ["$spent", "$budget"] } // Calculates the overspend as 'spent - budget'.
    }
  }
])
```

## $lookup

- Performs a left outer join between two collections
- used to combine data from multiple collections based on a specified relationship
- references documents from one collection within another collection

```
{
        $lookup: {
                from: "<foreign_collection>"
                localField: "<field_in_current_collection>",
                foreignField: "field_in_foreign_collection",
```

```
                as: "output_field",
        }
}
```

**from**: the name of the other collection you want to join
**localField**: the field from the current collection that you want to match
**foreignField**: the field from the collection that you want to match with localField
**as**: the name of the new array field where the matching documents from the collection will be stored

**Orders Collection**

```
[
  { "_id": 1, "product": "Pen", "quantity": 10, "customer_id": 101 },
  { "_id": 2, "product": "Notebook", "quantity": 5, "customer_id": 102 },
  { "_id": 3, "product": "Pencil", "quantity": 15, "customer_id": 101 }
]
```

**Customers Collection**

```
[
  { "_id": 101, "name": "Alice", "location": "New York" },
  { "_id": 102, "name": "Bob", "location": "Los Angeles" },
  { "_id": 103, "name": "Charlie", "location": "Chicago" }
]
```

**Query**

```
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customer_id",
      foreignField: "_id",
      as: "customer_details"
    }
  }
])
```

**Results**

```
[
  {
    "_id": 1,
    "product": "Pen",
    "quantity": 10,
    "customer_id": 101,
    "customer_details": [
      { "_id": 101, "name": "Alice", "location": "New York" }
    ]
  },
  {
    "_id": 2,
    "product": "Notebook",
    "quantity": 5,
    "customer_id": 102,
    "customer_details": [
      { "_id": 102, "name": "Bob", "location": "Los Angeles" }
    ]
  },
  {
    "_id": 3,
    "product": "Pencil",
    "quantity": 15,
    "customer_id": 101,
    "customer_details": [
      { "_id": 101, "name": "Alice", "location": "New York" }
```

```
    ]
  }
]
```

# Index

## Aggregation Expression: Operators

- `$abs`: Returns the absolute value of a number.
- `$accumulator`: Defines a custom aggregation function in the `$group` stage.
- `$acos`: Returns the arccosine (inverse cosine) of a number in radians.
- `$acosh`: Returns the inverse hyperbolic cosine (arccosh) of a number.
- `$add`: Adds numbers, dates, or arrays.
- `$addToSet`: Adds a value to a set, ensuring no duplicates in the `$group` stage.
- `$allElementsTrue`: Checks if all elements of an array evaluate to true.
- `$and`: Evaluates to true if all expressions evaluate to true.
- `$anyElementTrue`: Checks if any element of an array evaluates to true.
- `$arrayElemAt`: Returns the element at the specified index in an array.
- `$arrayToObject`: Converts an array into a single document.
- `$asin`: Returns the arcsine (inverse sine) of a number in radians.
- `$asinh`: Returns the inverse hyperbolic sine of a number.
- `$atan`: Returns the arctangent (inverse tangent) of a number in radians.
- `$atan2`: Returns the arctangent of two numbers (y and x coordinates).
- `$atanh`: Returns the inverse hyperbolic tangent of a number.
- `$avg`: Calculates the average of numeric values.
- `$binarySize`: Returns the size of a binary value in bytes.
- `$bsonSize`: Returns the size of a BSON document.
- `$ceil`: Rounds a number up to the nearest integer.
- `$cmp`: Compares two values and returns -1, 0, or 1.
- `$concat`: Concatenates strings.
- `$concatArrays`: Combines multiple arrays into a single array.
- `$cond`: Evaluates a conditional expression (if-then-else logic).
- `$convert`: Converts a value to a specified type.
- `$cos`: Returns the cosine of a number in radians.
- `$dateFromParts`: Constructs a date from individual date parts.
- `$dateFromString`: Converts a string to a date object.
- `$dateToParts`: Extracts date parts from a date object.
- `$dateToString`: Formats a date as a string.
- `$dayOfMonth`: Returns the day of the month for a date.
- `$dayOfWeek`: Returns the day of the week for a date.
- `$dayOfYear`: Returns the day of the year for a date.
- `$degreesToRadians`: Converts degrees to radians.
- `$divide`: Divides one number by another.
- `$eq`: Checks if two values are equal.
- `$exp`: Returns e (Euler's number) raised to the power of a specified number.
- `$filter`: Filters elements from an array.
- `$first`: Returns the first element in an array.
- `$floor`: Rounds a number down to the nearest integer.
- `$function`: Defines a custom function to execute in aggregation.
- `$gt`: Checks if a value is greater than another value.

- `$gte`: Checks if a value is greater than or equal to another value.
- `$hour`: Extracts the hour from a date object.
- `$ifNull`: Returns a specified value if the input is null or missing.
- `$in`: Checks if a value is in an array.
- `$indexOfArray`: Returns the index of an element in an array.
- `$indexOfBytes`: Returns the byte index of a substring in a string.
- `$indexOfCP`: Returns the code point index of a substring in a string.
- `$isArray`: Checks if the input is an array.
- `$isNumber`: Checks if the input is a number.
- `$isoDayOfWeek`: Returns the ISO day of the week for a date.
- `$isoWeek`: Returns the ISO week of the year for a date.
- `$isoWeekYear`: Returns the ISO week-numbering year for a date.
- `$last`: Returns the last element in an array.
- `$let`: Assigns variables for use in an aggregation pipeline.
- `$literal`: Returns a value without parsing or interpretation.
- `$ln`: Returns the natural logarithm of a number.
- `$log`: Returns the logarithm of a number for a specified base.
- `$log10`: Returns the base-10 logarithm of a number.
- `$lt`: Checks if a value is less than another value.
- `$lte`: Checks if a value is less than or equal to another value.
- `$ltrim`: Removes leading whitespace or characters from a string.
- `$map`: Applies an expression to each element in an array.
- `$max`: Returns the maximum value in a set of values.
- `$mergeObjects`: Combines multiple documents into a single document.
- `$meta`: Accesses metadata in a query.
- `$millisecond`: Extracts the millisecond from a date object.
- `$min`: Returns the minimum value in a set of values.
- `$minute`: Extracts the minute from a date object.
- `$mod`: Returns the remainder of a division operation.
- `$multiply`: Multiplies two or more numbers.
- `$ne`: Checks if two values are not equal.
- `$not`: Negates a specified expression.
- `$objectToArray`: Converts a document into an array of key-value pairs.
- `$or`: Evaluates to true if any expression evaluates to true.
- `$pow`: Raises a number to a specified power.
- `$push`: Appends a value to an array in the `$group` stage.
- `$radiansToDegrees`: Converts radians to degrees.
- `$range`: Generates an array of sequential numbers.
- `$reduce`: Applies an expression to reduce an array to a single value.
- `$regexFind`: Finds a match for a regular expression in a string.
- `$regexFindAll`: Finds all matches for a regular expression in a string.
- `$regexMatch`: Checks if a string matches a regular expression.
- `$replaceOne`: Replaces the first match of a string or regex with a specified value.
- `$replaceAll`: Replaces all matches of a string or regex with a specified value.
- `$reverseArray`: Reverses the order of elements in an array.
- `$round`: Rounds a number to a specified decimal place.
- `$rtrim`: Removes trailing whitespace or characters from a string.
- `$second`: Extracts the second from a date object.
- `$setDifference`: Returns elements in one array but not in another.

- `$setEquals`: Checks if two arrays contain the same elements.
- `$setIntersection`: Returns elements common to all input arrays.
- `$setIsSubset`: Checks if one array is a subset of another array.
- `$setUnion`: Combines elements from all input arrays without duplicates.
- `$sin`: Returns the sine of a number in radians.
- `$size`: Returns the size of an array.
- `$slice`: Extracts a subset of an array.
- `$split`: Splits a string into an array of substrings.
- `$sqrt`: Returns the square root of a number.
- `$stdDevPop`: Calculates the population standard deviation of numeric values.
- `$stdDevSamp`: Calculates the sample standard deviation of numeric values.
- `$strLenBytes`: Returns the number of bytes in a string.
- `$strLenCP`: Returns the number of UTF-8 code points in a string.
- `$strcasecmp`: Performs a case-insensitive string comparison.
- `$substr`: Extracts a substring from a string.
- `$substrBytes`: Extracts a substring based on byte indexes.
- `$substrCP`: Extracts a substring based on UTF-8 code point indexes.
- `$sum`: Calculates the sum of numeric values.
- `$switch`: Evaluates multiple conditions and returns a value for the first true condition.
- `$tan`: Returns the tangent of a number in radians.
- `$toBool`: Converts a value to a boolean.
- `$toDate`: Converts a value to a date.
- `$toDecimal`: Converts a value to a decimal.
- `$toDouble`: Converts a value to a double.
- `$toInt`: Converts a value to an integer.
- `$toLong`: Converts a value to a 64-bit integer.
- `$toLower`: Converts a string to lowercase.
- `$toObjectId`: Converts a value to an ObjectId.
- `$toString`: Converts a value to a string.
- `$toUpper`: Converts a string to uppercase.
- `$trim`: Removes leading and trailing whitespace or characters from a string.
- `$trunc`: Truncates a number to a whole number or specified decimal place.
- `$type`: Returns the BSON type of a field.
- `$week`: Returns the week of the year for a date.
- `$year`: Returns the year of a date.
- `$zip`: Combines arrays into a single array of subarrays.